> "Its very illuminating to think about the fact that some — at most four hundred — years ago, professors at European universities would tell the brilliant students that if they were very diligent, it was not impossible to learn how to do long division. You see, the poor guys had to do it in Roman numerals. Now, here you see in a nutshell what a difference there is in a good and bad notation."
>
> – Edsger W. Dijkstra
> *Datamation* Vol.23, No.5, p.164, 1977

Lecture II
RECURRENCES

Recurrences arise naturally in the complexity analysis of recursive algorithms and in probabilistic analysis. We introduce some basic techniques for solving recurrences. A recurrence is a recursive relation for a complexity function $T(n)$. Here are two examples:

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

and

$$T(n) = n + 2T(n/2). \tag{2}$$

Looks familiar...

The reader may recognize the first as the recurrence for Fibonacci numbers, and the second as the complexity of the Mergesort, described in Lecture 1. These recurrences have[1] the following "separable form":

$$T(n) = G(n, T(n_1), \ldots, T(n_k)) \tag{3}$$

where $G(x_0, x_1, \ldots, x_k)$ is a function in $k+1$ variables and each n_i ($i = 1, \ldots, k$) is a function of n that is strictly less than n. E.g., in (1), we have $k = 2$ and $n_1 = n-1, n_2 = n-2$ while in (2), we have $k = 1$ and $n_1 = n/2$.

What does it mean to "solve" recurrences such as equations (1) and (2)? The Fibonacci and Mergesort recurrences have the following well-known solutions:

$$F(n) = \Theta(\phi^n)$$

where $\phi = (1+\sqrt{5})/2 = 1.618\ldots$ is the golden ratio, and

Solve up to Θ-order

$$T(n) = \Theta(n \log n).$$

In this book, we generally estimate complexity functions $T(n)$ only up to its Θ-order. If only an upper bound or lower bound is needed, and we determine $T(n)$ up to its O-order or to Ω-order. In rare cases, we may be able to derive the exact solution (in fact, this is possible for $T(n)$ and $F(n)$ above). One benefit of Θ-order solutions is this — most of the recurrences we treat in this book can be solved by only elementary methods, without assuming differentiability or using calculus tools.

The variable "n" is called the **designated variable** of the recurrence (3). If there are non-designated variables, they are supposed to be held constant. In mathematics, we usually reserve "n" for natural numbers or perhaps integers. In the above examples, this is the natural interpretation for n. But one of the first steps we take in solving recurrences is to re-interpret n (or whatever is the designated variable) to range over the real numbers. The corresponding recurrence equation (3) is then called a **real recurrence**. For this reason, we may prefer the symbol "x" as our designated variable, since x is normally viewed

All recurrences are real

[1] Non-separable recurrences looks like $G(n, T(n), T(n_1), \ldots, T(n_k)) = 0$, but these are rare.

as a real variable.

What does an extension to real numbers mean? In the Fibonacci recurrence (1), what is $F(2.5)$? In Mergesort (2), what does $T(\pi) = T(3.14159\ldots)$ represent? The short answer is, we don't really care.

In addition to the recurrence (3), we generally need the **boundary conditions** or **initial values** of the function $T(n)$. They give us the values of $T(n)$ *before* the recurrence (3) becomes valid. Without initial values, $T(n)$ is generally under-determined. For our example (1), if n ranges over natural numbers, then the initial conditions

$$F(0) = 0, \qquad F(1) = 1$$

Some initial conditions lead to trivial solutions

give rise to the standard Fibonacci numbers, *i.e.*, $F(n)$ is the nth Fibonacci number. Thus $F(2) = 1, F(3) = 2, F(4) = 3$, etc. On the other hand, if we use the initial conditions $F(0) = F(1) = 0$, then the solution is trivial: $F(n) = 0$ for all $n \geq 0$. Thus, our assertion earlier that $F(n) = \Theta(\phi^n)$ is the solution to (1) is not[2] really true without knowing the initial conditions. On the other hand, $T(n) = \mathcal{O}(n \log n)$ can be shown to hold for (2) regardless of the initial conditions. For the typical recurrence from complexity analysis, this will be the case.

EXERCISES

Exercise 0.1: Consider the non-homogeneous version of Fibonacci recurrence $F(n) = F(n-1) + F(n-2) + f(n)$ for some function $f(n)$. If $f(n) = 1$, show that $F(n) = \Omega(c^n)$ for some $c > 1$, regardless of the initial conditions. Try to find the largest value for c. Does your bound hold if we have $f(n) = n$ instead? ♢

Exercise 0.2: Let $T(n) = aT(n/b) + n$, where $a > 0$ and $b > 1$. How sensitive is this recurrence to the initial conditions? More precisely, if $T_1(n)$ and $T_2(n)$ are two solutions corresponding to two initial conditions, what is the strongest relation you can infer between T_1 and T_2? ♢

Exercise 0.3: (Aho and Sloane, 1973) Consider recurrences of the form

$$T(n) = (T(n-1))^2 + g(n). \tag{4}$$

For this exercise, we assume n is a natural numbers and use explicit boundary conditions.
(a) Show that the number of binary trees of height at most n is given by this recurrence with $g(n) = 1$ and the boundary condition $T(1) = 1$. Show that this particular case of (4) has solution

$$T(n) = \left\lfloor k^{2^n} \right\rfloor. \tag{5}$$

(b) Show that the number of Boolean functions on n variables is given by (4) with $g(n) = 0$ and $T(1) = 2$. Solve this. ♢

Exercise 0.4: Let T, T' be binary trees and $|T|$ denote the number of nodes in T. Define the relation $T \sim T'$ recursively as follows: (BASIS) If $|T| = 0$ or 1 then $|T| = |T'|$. (INDUCTION) If $|T| > 1$ then $|T'| > 1$ and either (i) $T_L \sim T'_L$ and $T_R \sim T'_R$, or (ii) $T_L \sim T'_R$ and $T_R \sim T'_L$. Here T_L and T_R denote the left and right subtrees of T.
(a) Use this to give a recursive algorithm for checking if $T \sim T'$.
(b) Give the recurrence satisfied by the running time $t(n)$ of your algorithm.
(c) Give asymptotic bounds on $t(n)$. ♢

[2] The reason behind this is that (1) is a homogeneous recurrence while (2) is non-homogeneous. For instance, $F(n) = F(n-1) + F(n-2) + 1$ would be non-homogeneous and its Θ-solution would not depend on the initial conditions.

Exercise 0.5: Let $\phi = (1+\sqrt{5})/2 \approx 1.618$ and $\widehat{\phi} = (1-\sqrt{5})/2 \approx -0.618$. If $F(n)$ satisfies the Fibonacci recurrence $F(n) = F(n-1) + F(n-2)$, we said in the text that $F(n) = \Theta(\phi^n)$. Let us now give the exact solution for this recurrence.
(a) Use induction to show that $F(n) = \phi^n/\sqrt{5} - \widehat{\phi}^n/\sqrt{5}$ is the solution with the initial conditions $F(n) = n$ for $n = 0, 1$.
(b) Some authors like to begin with $F(n) = 1$ for $n = 0, 1$. Find the constants a, b such that $F(n) = a\phi^n + b\widehat{\phi}^n$ for all $n \in \mathbb{N}$.
(c) In general, how can you give an exact formula for $F(n)$ given that you know the value of $F(n)$ at two consecutive values of n (say $n = n_0$ and $n = n_0 + 1$)? Is it strictly necessary for $n_0 = 0$, and ♢

END EXERCISES

§1. Simplification

In the real world, when faced with an actual recurrence to be solved, there are usually some simplifications steps to be taken. Here are three general simplifications that should be automatically taken:

- **Initial Condition.** In this book, we normally state recurrence without any initial conditions. This is deliberate: we expect the student to supply the initial conditions, based on the following assumption: how convenient!

  > **Default Initial Condition** (DIC): *There is some $n_1 > 0$ such that for all $n < n_1$, $T(n)$ is assigned arbitrary values. The recurrence for $T(n)$ holds for all $n \geq n_1$.*

  The intent is for the student to make convenient choices for n_1 and the initial values of $T(n)$. Normally, we make choices so that the resulting solution has a simple form. Our favorite version of DIC is $T(n) = C$ for all $n < n_1$ and some constant C. To use DIC, we need not specify n_1 or the initial values of $T(n)$ before hand. We just proceed to solve the recurrence, and at the appropriate moments, introduce these values.

  What is the justification for this approach? It allows us to focus on the recurrence itself rather than the initial conditions. In many cases, this arbitrariness does not affect the asymptotic behavior of the solution. Even if this simplification is not valid, we might have learned something about the recurrence.

- **Extension to Real Functions.** Even if the function $T(n)$ is originally defined for natural numbers n, we will now treat $T(n)$ as a real function (*i.e.*, n is viewed as a real variable), and defined for n sufficiently large. See the Exercise for a standard approach ("ample domain") that avoids extensions to real functions. It is important to realize that even if we have no interest in real recurrences, some solution techniques below will transform our recurrences into non-integer recurrences. So we might as well take the plunge from the start. But the best recommendation for this approach is its simplicity and naturalness.

- **Converting Recurrence Inequality into a Recurrence Equation.** If we begin with a recurrence inequality such as $T(n) \leq G(n, T(n_1), \ldots, T(n_k))$, we simply rewrite this as an equality relation: $T(n) = G(T(n_1), \ldots, T(n_k))$. Because of this change, our eventual solution for $T(n)$ is only an upper bound on the original function. Similarly, if we had started with $T(n) \geq G(n, T(n_1), \ldots, T(n_k))$, the eventual solution is only a lower bound.

¶1. Special Simplifications. Suppose the running time of an algorithm satisfies the following inequality:

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 6n + \lg n - 4, \tag{6}$$

for integer $n > 100$, with boundary condition

$$T(n) = 3n^2 - 4n + 2 \tag{7}$$

for $0 \leq n \leq 100$. Such a **recurrence in-equation** may arises in some imagined implementation of Mergesort, with special treatment for $n \leq 100$. Our general simplification steps tells us to (a) discard the specific boundary conditions (7) in favor of DIC, (b) treat $T(n)$ as a real function, and (c) write the recurrence as a equation.

What other simplifications might apply here? Let us convert (6) into the following

$$T(n) = 2T(n/2) + n. \tag{8}$$

This represents two additional simplifications: (i) We replaced the term "$+6n + \lg n - 4$" by some simple expression ("$+n$") with same Θ-order. (ii) We have removed the ceiling and floor functions. Step (i) is justified because this does not affect the Θ-order (if this is not clear, then you can always come back to verify this claim). Step (ii) exploits the fact that we now treat $T(n)$ as a real function, so we need not worry about non-integral arguments when we remove the ceiling or floor functions. Also, it does not affect the asymptotic value of $T(n)$ here.

The justifications for these steps are certainly not obvious, but they should seem reasonable. Ultimately, one ought to return to such simplifications to justify them.

EXERCISES

Exercise 1.1: Show that our above simplifications of the the recurrence (6) (with its initial conditions) cannot affect the asymptotic order of the solution. [Show this for ANY choice of Default Initial Condition.] ♢

Exercise 1.2: We seek counterexamples to the claim that we can replace $\lceil n/2 \rceil$ by $n/2$ in a recurrence without changing the Θ-order of the solution.
(a) Construct a function $g(n)$ that provides a counter example for the following recurrence: $T(n) = T(\lceil n/2 \rceil) + g(n)$. HINT: make $g(n)$ depend on the parity of n.
(b) Construct a different counter example of the form $T(n) = h(n)T(\lceil \frac{n}{2} \rceil)$ for a suitable function $h(n)$. ♢

Exercise 1.3: Show examples where the choice of initial conditions can change the Θ-order of the solution $T(n)$. HINT: Choose $T(n)$ to increase exponentially. ♢

Exercise 1.4: Suppose x, n are positive numbers satisfying the following "non-separable recurrence" equation,

$$2^x = x^{2n}.$$

Solve for x as a function of n, showing

$$x(n) = [1 + o(1)]2n \log_2(2n).$$

HINT: take logarithms. This is an example of a bootstrapping argument where we use an approximation of $x(n)$ to derive yet a better approximation. See, e.g., Purdom and Brown [16]. ♢

Exercise 1.5: [Ample Domains] Our approach of considering real functions is non-standard. The standard approach to solving recurrences in the algorithms literature is the following. Consider the simplification of (6) to (8). Suppose, instead of assuming $T(n)$ to be a real function (so that (8) makes sense for all values of n), we continue to assume n is a natural number. It is easy to see that $T(n)$ is completely defined by (8) iff n is a power of 2. We say that (8) is closed over the set $D_0 := \{2^k : k \in \mathbb{N}\}$ of powers of 2. In general, we say a recurrence is "closed over a set $D \subseteq \mathbb{R}$" if for all $n \in D$, the recurrence for $T(n)$ depends only on smaller values n_i that also belong in D (unless n_i lies within the boundary condition).
(a) Let us call a set $D \subseteq \mathbb{R}$ an "ample set" if, for some $\alpha > 1$, the set $D \cap [n, \alpha \cdot n]$ is non-empty for all $n \in \mathbb{N}$. Here $[n, \alpha n]$ is closed real interval between n and αn. If the solution $T(n)$ is sufficiently "smooth", then knowing the values of $T(n)$ at an ample set D gives us a good approximation to values where $n \notin D$. In this question, our "smoothness assumption" is simply: $T(n)$ *is monotonic non-decreasing.* Suppose that $T(n) = n^k$ for n ranging over an ample set D. What can you say about $T(n)$ for $n \notin D$? What if $T(n) = c^n$ over D? What if $T(n) = 2^{2^n}$ over D?
(b) Suppose $T(n)$ is recursively expressed in terms of $T(n_1)$ where $n_1 < n$ is the largest prime smaller than n. Is this recurrence defined over an ample set? ♢

Exercise 1.6: Consider inversions in a sequence of numbers.
(a) The sequence $S_0 = (1, 2, 3, 4)$ has no inversions, but sequence $S_1 = (2, 1, 4, 3)$ has two inversions, namely the pairs $\{1, 2\}$ and $\{3, 4\}$. Now, the sequence $S_2 = (2, 3, 1, 4)$ also has two inversions, namely the pairs $\{1, 2\}$ and $\{1, 3\}$. Let $I(S)$ be the number of inversions in S. Give an $O(n \lg n)$ algorithm to compute $I(S)$. Hint: this is a generalization of Mergesort.
(b) We next distinguish between the quality of the inversions of S_1 and S_2. The inversions $\{1, 2\}$ and $\{3, 4\}$ in S_1 are said to have weight of 1 each, so the **weighted inversion** of S_1 is $W(S_1) = 2 = 1 + 1$. But for S_2, the inversion $\{1, 2\}$ has weight 2 while inversion $\{1, 3\}$ has weight 1. So the weighted inversion is $W(S_2) = 3 = 2 + 1$. Thus the "weight" measures how far apart the two numbers are. In general, if $S = (a_1, \ldots, a_n)$ then a pair $\{a_i, a_j\}$ is an **inversion** if $i < j$ and $a_i > a_j$. The weight of this inversion is $j - i$. Let $W(S)$ be the sum of the weights of all inversions. Give an $O(n \lg n)$ algorithm for weighted inversions. ♢

Exercise 1.7: We might consider following form of DIC where we assume that there exists $0 < n_0 < n_1$, and constants $0 < C_0 \leq C_1$ such that

$$(\forall\, n_0 \leq n < n_1)[C_0 \leq T(n) \leq C_1]. \tag{9}$$

Solve the Fibonacci and mergesort recurrences using this version of DIC. Your solutions should be stated in terms of the parameters C_1, C_2. ♢

END EXERCISES

§2. Divide-and-Conquer Algorithms

In this section, we see some other interesting recurrences that arise in a divide-and-conquer algorithms. First, we look at Karatsuba's classic algorithm for multiplying integers [10]. Then we consider a modern problem arising in searching for key words.

OK, you learned it in *grade* school

¶2. Example from Arithmetic. To motivate Karatsuba's algorithm, let us recall the classic "high-school algorithm" for multiplying integers. Given positive integers X, Y, we want to compute their product $Z = XY$. This algorithm assumes you know how to do single-digit multiplication and multi-digit additions ("pre-high school"). The algorithm multiples X by each digit of Y. If X and Y have n digits each, then we now have n products, each having at most $n+1$ digits. After appropriate left-shifts of these n products, we add them all up. It is not hard to see that this algorithm takes $\Theta(n^2)$ time. Can we improve on this?

Recall the introductory remark from Dijkstra on the importance of notation for algorithms

Usually we think of X, Y in decimal notation, but the algorithm works equally well in any base. We shall assume base 2 for simplicity. For instance, if $X = 19$ then in binary $X = 10011$. To avoid the ambiguity from different bases, we indicate[3] the base using a subscript, $X = (10011)_2$. The standard convention is that decimal base is assumed when no base is indicated. Thus a plain "100" without any base represents one hundred, but $(100)_2$ represents four.

Assume X and Y has length exactly n where n is a power of 2 (we can pad with 0's if necessary). Let us split up X into a high-order half X_1 and low-order half X_0. Thus

$$X = X_0 + 2^{n/2} X_1$$

where X_0, X_1 are $n/2$-bit numbers. Similarly,

$$Y = Y_0 + 2^{n/2} Y_1.$$

Then

$$\begin{aligned} Z &= (X_0 + 2^{n/2} X_1)(Y_0 + 2^{n/2} Y_1) \\ &= X_0 Y_0 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + 2^n X_1 Y_1 \\ &= Z_0 + 2^{n/2} Z_1 + 2^n Z_2, \end{aligned}$$

where $Z_0 = X_0 Y_0$, etc. Clearly, each of these Z_i's have at most $2n$ bits. Now, if we compute the 4 products

$$X_0 Y_0, X_1 Y_0, X_0 Y_1, X_1 Y_1$$

recursively, then we can put them together ("conquer step") in $\mathcal{O}(n)$ time. To see this, we must make an observation: in binary notation, multiplying any number X by 2^k (for any positive integer k) takes $\mathcal{O}(k)$ time, independent of X. We can view this as a matter of shifting left by k, or by appending a string of k zeros to X.

Hence, if $T(n)$ is the time to multiply two n-bit numbers, we obtain the recurrence

$$T(n) \leq 4T(n/2) + Cn \tag{10}$$

for some $C > 1$. Given our simplification suggestions, we immediately rewrite this as

$$T(n) = 4T(n/2) + n.$$

As we will see, this recurrence has solution $T(n) = \Theta(n^2)$, so we have not really improved on the high-school method.

Karatsuba observed that we can proceed as follows: we can compute $Z_0 = X_0 Y_0$ and $Z_2 = X_1 Y_1$ first. Then we can compute Z_1 using the formula

$$Z_1 = (X_0 + X_1)(Y_0 + Y_1) - Z_0 - Z_2.$$

[3] By the same token, we may write $X = (19)_{10}$ for base 10. But now the base "10" itself may be ambiguous — after all "10" in binary is equal to two. The convention is to write the base in decimal.

Thus Z_1 can be computed with one recursive multiplication plus some additional $\mathcal{O}(n)$ work. From Z_0, Z_1, Z_2, we can again obtain Z in $\mathcal{O}(n)$ time. This gives us the **Karatsuba recurrence**,

$$T(n) = 3T(n/2) + n. \tag{11}$$

first improvement in 1000 years? According to Wikipedia, high school multiplication is equivalent to the "lattice method" which is at least 1000 years old.

We shall show that $T(n) = \Theta(n^\alpha)$ where $\alpha = \lg 3 = 1.58\cdots$. This is clearly an improvement of the high school method.

There is an even faster algorithm from Schönhage and Strassen (1971) that runs in $O(n \log n \log\log n)$ time. This has withstood improvements for almost 20 years, but in recent years, the $\log\log n$ factor has begun to be breached (they can be replaced by $\log^* n$). Many theoretical computer scientists believe that an $O(n \log n)$ algorithm should be possible. There is an increasing need for multiplication of arbitrarily large integers. In cryptography or computational number theory, for example. These are typically implemented in software in a "big integer" package. For instance, `Java` has a `BigInteger` class. A well-engineered big integer multiplication algorithm will typically implement the High-School algorithm for $n \leq n_0$, and use Karatsuba for $n_0 < n \leq n_1$, and use Schönhage-Strassen for $n > n_1$. Typical values for n_0, n_1 are $30, 200$.

¶3. A Google Problem. The Google Phenomenon is possible because of efficient algorithms: every files on the web can be searched and indexed. Searching is by keywords. Let us suppose that Google pre-processes every file in its database for keywords. However, a user may ask to search files for two or more keywords. We will reduce this multi-keyword search to a precomputed single-keyword index.

Let F be a file, viewed as a sequence of words (ignoring punctuation, capitalization, etc). We first pre-process F for the occurrences of keywords. For each keyword w, we precompute an **index** which amounts a sorted sequence $P(w)$ of positions indicating where w occurs in F. E.g.,

$$P(divide) = (11, 16, 42, 101, 125, 767)$$

means that the keyword $divide$ occurs 6 times in F, at positions $11, 16$, etc. Suppose we want to search the file using a conjunction of k keywords, $w_1, \ldots, w_k$. An interval $J = [s, t]$ is called a **cover** for $w_1, \ldots, w_k$ if each w_i occurs at least once within the positions in J. The size of a cover $[s, t]$ is just $t - s$. A cover is **minimal** if it is not contained in some larger cover; it is **minimum** if its size is smallest among all covers. Note that if $[s_i, t_i]$ are minimal covers for $i = 1, 2, \ldots$, and if $s_i < s_{i+1}$ then $t_i < t_{i+1}$. The **keyword cover problem** is this: given the indices $P(w_1), \ldots, P(w_k)$ for a set $W = \{w_1, \ldots, w_k\}$ of keywords in a file, to compute a minimum cover for W.

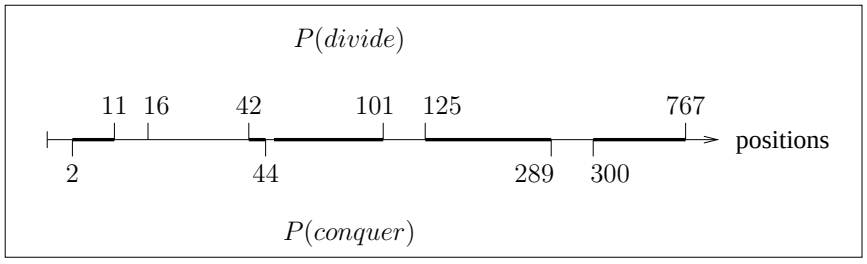


Figure 1: Minimal Covers

E.g., let $k = 2$ with $w_1 = divide$ and $w_2 = conquer$. With $P(divide)$ as before, let $P(conquer) = (2, 44, 289, 300)$. Then the minimal covers are $[2, 11], [42, 44], [44, 101], [125, 289], [289, 767]$. This is illustrated in Figure 1. The minimum cover is $[42, 44]$.

Before attempting to solve this problem, consider how Google might use the minimum cover solutions: suppose a user wants to search for a set $W = \{w_1, \ldots, w_k\}$ of key words. For each file f_j ($j = 1, 2, \ldots$) we use the algorithm to compute a minimum cover $[c_j, d_j]$ (if one exists) for W in f_j. The indices $P(w_i)$ for each key word w_i are assumed to have been precomputed. The search results will be a list of all files for which covers exist, but we order these files in order of non-decreasing cover size $d_j - c_j$. The actual cover $[c_j, d_j]$ can be used by Google to display a snippet of the file f_j.

Let us now consider algorithms. Let n_i be the length of list $P(w_i)$ ($i = 1, \ldots, k$) and $n = n_1 + \cdots + n_k$. The case $k = 2$ is relatively straightforward, and we leave it for an exercise. Consider the case $k = 3$. First, merge $P(w_1), P(w_2), P(w_3)$ into the array $A[1..n]$. Recall that in Lecture I, we discussed the merging of sorted lists. Merging takes time $O(n_1 + n_2 + n_3) = O(n)$. To keep track of the origin of each number in A, we may also construct an array $B[1..n]$ such that $B[i] = j \in \{1, 2, 3\}$ iff $A[i]$ comes from the list $P(w_j)$.

We use a divide-and-conquer approach. Recursively, compute a minimum cover of $A[1..(n/2)]$ and $A[(n/2) + 1..n]$ (for simplicity, assume n is a power of 2). Let $C_{1,n/2}$ and $C_{(n/2)+1,n}$ be these minimum covers. We now need to find a minimal cover that straddles $A[(n/2)]$ and $A[(n/2) + 1]$. Let $C = [A[i], A[j]]$ be such a minimal cover, where $i \leq (n/2)$ and $j \geq (n/2) + 1$. There are 6 cases. One case is when $C = C' \cup C''$, where $C' = [A[i], A[n/2]]$ is the rightmost cover for w_1 in $A[1..(n/2)]$, and $C'' = [A[(n/2) + 1], A[j]]$ is the leftmost cover for w_2, w_3 in $A[(n/2) + 1, n]$. We can find C' and C'' in $O(n)$ time. The remaining 5 cases can similarly be found in $O(n)$ time. Then C is the cover that has minimum size among these 6 cases. Hence, the overall complexity of the algorithm satisfies

$$T(n) = 2T(n/2) + n.$$

We have seen this recurrence before, as the Mergesort recurrence (2). The solution is $T(n) = \Theta(n \log n)$. See exercise for a general solution in $O(n \log k)$ time.

¶4. Master Recurrence and Divide-and-Conquer Algorithms. The recurrences (2) and (11) are instances of the **Master Recurrence** which has the form:

$$T(n) = aT(n/b) + d(n) \tag{12}$$

where $a > 0$ and $b > 1$ are constants and d is any function, usually called the **driving** or **forcing function**. Below, we shall solve this recurrence under fairly general conditions.

The idea of solving a problem by reducing it to smaller subproblems is a very general one. In this chapter, we mainly focus on reductions from problems of size n to subproblems of size $\leq cn$ for some fixed $c < 1$. If there are a finite number of such subproblems, the running times can be bounded using solutions to the Master recurrence (12). In other problems, we reduce a problem of size n to several subproblems that of size $\leq n - c$ for some fixed $c \geq 1$. Such solutions would be exponential time without additional properties; we study these under the topic of dynamic programming (Chapter 7). In applications, we have $d(n) > 0$, representing the cost of merging solutions of subproblems in divide-and-conquer algorithms.

EXERCISES

Exercise 2.1: Carry out Karatsuba's algorithm for $X = 6 = (0110)_2$ and $Y = 11 = (1011)_2$. It is enough to display the recursion tree with the correct arguments for each recursive call, and the returned values. ♢

NumBits	AvgTime	Exponent	NumBits	AvgTime	Exponent
4000	4.358	0.0	9600	23.034	1.9017905239616146
4200	4.696	1.531002145103799	9800	24.055	1.9064306092855452
4400	5.194	1.841260577604784	10000	24.986	1.905838802838669
4600	5.517	1.6873048110254347	10200	25.987	1.9074840762036238
4800	5.983	1.7381865504999572	10400	26.948	1.9067232067781992
5000	6.51	1.7985113947251763	10600	28.108	1.912700793571853
5200	6.988	1.7997159663026001	10800	29.111	1.9120055203582398
5400	7.509	1.812998128928515	11000	30.221	1.9143159996069712
5600	8.01	1.8089977665618309	11200	31.534	1.922120988851413
5800	8.684	1.85558837393382	11400	31.542	1.8898795547030012
6000	9.183	1.838236378924439	11600	32.67	1.8920105894497778
6200	9.769	1.8418523402197153	11800	33.703	1.8908891117429292
6400	10.365	1.8434357852847953	12000	34.67	1.8877101089855162
6600	11.088	1.864808884276074	12200	36.082	1.8955269064390694
6800	11.717	1.8638802969571109	12400	37.218	1.8956825843907563
7000	12.413	1.8704459319724756	12600	38.049	1.8884930574030907
7200	13.092	1.8714070696035303	12800	39.242	1.8894663931349043
7400	13.843	1.8787279477010768	13000	40.553	1.892493164635265
7600	14.532	1.8763458534440565	13200	41.696	1.8915733844170872
7800	15.297	1.8801860861195574	13400	42.951	1.8925738155123988
8000	16.054	1.8811947011507577	13600	44.159	1.8923271871808227
8200	16.905	1.8884383570994894	13800	45.533	1.8947617307075215
8400	17.644	1.8847717474449632	14000	46.816	1.8951803717241376
8600	18.498	1.8885827751677746	14200	48.1	1.8953182704475686
8800	19.283	1.8862283707110576	14400	49.401	1.8954588786790316
9000	20.225	1.8927722703240168	14600	50.873	1.8979435636574864
9200	21.17	1.8976522229154338	14800	52.364	1.9002856600816482
9400	22.063	1.8982439890258536	15000	53.537	1.8977482007273088

Figure 2: Timing as a function of number of bits

Exercise 2.2: Suppose an implementation of Karatsuba's algorithm achieves $T(n) \leq Cn^{1.58}$ where $C = 1000$. Moreover, the High School multiplication is $T(n) = 30n^2$. Beyond what value of n does Karatsuba definitely becomes competitive with the High School method? ♢

Exercise 2.3: Consider the recurrence $T(n) = 3T(n/2) + n$ and $T'(n) = 3T'(\lceil n/2 \rceil) + 2n$. Show that $T(n) = \Theta(T'(n))$. ♢

Exercise 2.4: The following is a programming exercise. It is best done using a programming language such as Java that has a readily available library of big integers.

(a) Implement Karatsuba's algorithm using such a programming language and using its big integer data structures and related facilities. The only restriction is that you must not use the multiplication, squaring, division or reciprocal facility of the library. But you are free to use its addition/subtraction operations, and any ability to perform left/right shifts (multiplication by powers of 2).

(b) Let us measure the running time of your implementation of Karatsuba's algorithm. For input numbers, use a random number generator to produce numbers of any desired bit length. If $T(n) \leq Cn^\alpha$ then $\lg T(n) \leq \lg C + \alpha \lg n$. The **exponent** α is thus the slope of the curve obtained by plotting $\lg T(n)$ against $\lg n$, we should get a slope of at most α. Plot the running time of your implementation to verify that its exponent is < 1.58.

(c) What is the exponent in Java's native implementation? Explain your data.

(d) My 1999 undergraduate class in algorithms did the preceding exercise, using the `java.math.BigInteger` package. One timing from this class is shown in Table 2. The "exponent" in this table is computing with a crude formula $\frac{\lg(avgTime) - avgTime_0}{\lg(numBits) - numBits_0}$ where $numBits_0 = 4000$ and $avgTime_0 = 4.358$ (the initial trial). This crude exponent hovers around 1.9. What would be the empirical exponent if you do a proper regression analysis? This data suggests that in 1999, the library only implemented the High School algorithm. By 2001, the situation appeared to have improved. ♢

Exercise 2.5: Suppose the running time of an algorithm is an unknown function of the form $T(n) = An^a + Bn^b$ where $a > b$ and A, B are arbitrary positive constants. You want to discover the exponent a by measurement. How can you, by plotting the running time of the algorithm for various n, find a with an error of at most ϵ? Assume that you can do least squares line fitting. ♢

Exercise 2.6: Try to generalize Karatsuba's algorithm by breaking up each n-bit number into 3 parts. What recurrence can you achieve in your approach? Does your recurrence improve upon Karatsuba's exponent of $\lg 3 = 1.58\cdots$? ♢

Exercise 2.7: To generalize Karatsuba's algorithm, consider splitting an n-bit integer X into m equal parts (assuming m divides n). Let the parts be $X_0, X_1, \ldots, X_{m-1}$ where $X = \sum_{i=0}^{m-1} X_i 2^{in/m}$. Similarly, let $Y = \sum_{i=0}^{m-1} Y_i 2^{in/m}$. Let us define $Z_i = \sum_{j=0}^{i} X_j Y_{i-j}$ for $i = 0, 1, \ldots, 2m-2$. In the formula for Z_i, assume $X_\ell = Y_\ell = 0$ when $\ell \geq m$.
(i) Determine the Θ-order of $f(m, n)$, defined to be the time to compute the product $Z = XY$ when you are given $Z_0, Z_1, \ldots, Z_{2m-2}$. Remember that $f(m, n)$ is the number of bit operations.
(ii) It is known that we can compute $\{Z_0, Z_1, \ldots, Z_{2m-2}\}$ from the X_i's and Y_j's using $\mathcal{O}(m \log m)$ multiplications and $\mathcal{O}(m \log m)$ additions, all involving (n/m)-bit integers. Using this fact with part (i), give a recurrence relations for the time $T(n)$ to multiply two n-bit integers.
(iii) Conclude that for every $\varepsilon > 0$, there is an algorithm for multiplying any two n-bit integers in time $T(n) = \Theta(n^{1+\varepsilon})$. NOTE: part (iii) is best attempted after you have studied the Master Theorem in the subsequent sections. ♢

Exercise 2.8: In the Google problem, we need to merge several sorted lists. Recall from Lecture I that we can merge a two lists of sizes m and n in time $\Theta(m + n)$. Suppose $X_1, \ldots, X_n$ are $n \geq 1$ sorted lists, each with $k \geq 1$ elements. Here, n and k are independent parameters.
(a) We want to analyze the complexity $T(n, k)$ of sorting the set $X = \bigcup_{i=1}^{n} X_i$. At each phase, we merge pairs of lists. With n lists of size k, we take $O(nk)$ time to merge, and produce $n/2$ lists each of size $2k$. Set up the recurrence for $T(n, k)$ based on this repeated merging algorithm.
(b) Show that $T(n, k) = \mathcal{O}(nk \lg(1 + n))$ (we say "$1 + n$" to ensure that the logarithm does not vanish when $n = 1$). HINT: you could use domain transformation (see §7) but this is not necessary.
(c) Use the Information Theoretic Lower Bound from Lecture I to show a lower bound of $\Omega(nk \lg(1 + n))$. ♢

Adapted from a Google interview question (the interviewed student was hired)

Exercise 2.9: Recall the Google multi-keyword search. This was reduced to computing a minimum cover for a set $W = \{w_1, \ldots, w_k\}$ of key words in a file. For each key word $w \in W$, we are given an index $P(w)$ which is just a sorted list of positions where w occurs in the file.
(a) Solve the minimum cover for $k = 2$ in linear time.
(b) Suppose $P(w_i) = (s_i, t_i)$ for each $i = 1, \ldots, k$, i.e., each keyword has just two positions. Give an $O(k \log k)$ algorithm to find the minimum cover C for $w_1, \ldots, w_k$. HINT: suppose the minimal covers are $C_1, \ldots, C_m$ for some $m \geq 1$. Give an algorithm to list all the minimal covers. If $C_i = [c_i, d_i]$ and assuming $c_1 < c_2 < \cdots < c_m$, how do you find C_1? How do you find C_{i+1} given C_i?
(c) Solve the general Google problem (k is arbitrary and each word can have arbitrarily many occurrences in the file). HINT: if you used the hint from (b), it should be possible to generalize your solution. ♢

Exercise 2.10: Write a program to solve the Google multi-keyword for the case $k = 3$ as described in the text. Use your favorite programming language (C or Java without any Object-Oriented fanfare

is recommended). Initially, assume n is a power of 2. Indicate how to adapt your algorithm when n is not a power of 2. ♢

Exercise 2.11: Consider the following problem: we are given an array $A[1..n]$ of numbers, possibly with duplicates. Let $f(x)$ be the number of times ("frequency") a number x occurs. Given a number $k \geq 1$, we want to know whether there are k distinct numbers $x_1, \ldots, x_k$ such that $\sum_{i=1}^{k} f(x_i) > n/2$. Call $\{x_1, \ldots, x_k\}$ a **k-majority set**.
(a) Solve this decision problem for $k = 1$.
(b) Solve this decision problem for $k = 2$.
(c) Instead of the previous decision problem, we consider the optimization version: find the smallest k such that there are k numbers $x_1, \ldots, x_k$ with $\sum_{i=1}^{k} f(x_i) > n/2$. ♢

END EXERCISES

§3. Rote Method

We are going to introduce two "direct methods" for solving recurrences: rote method and induction. They are "direct" as opposed to other transformation methods which we will introduce later. Although fairly straightforward, these direct methods may call for some creativity (educated guesses). We begin with the rote method, as it appears to require somewhat less guess work.

"...at last, a method named after me!" — Günter Rote

¶5. What is rote? The "rote method" refers to the idea of solving a recurrence by repeated expansion of a recurrence. Since such expansions can be done mechanically, this method has been characterized as rote.

Let us illustrate this method using the merge-sort recurrence (8): $T(n) = 2T(n/2) + n$. The important thing is that we can replace n in this by any expression: plugging $n/2$ for n in the recurrence, we get $T(n/2) = 2T(n/4) + n/2$. If we plug this back into the original recurrence, we get our second expansion in the following derivation:

$$\left.\begin{array}{rcll} T(n) & = & 2\boxed{\text{T(n/2)}} + n & \text{(first expansion)} \\ & = & 2\boxed{\text{2T(n/4)+(n/2)}} + n & \text{(second expansion)} \\ & = & 4\boxed{\text{T(n/4)}} + 2n & \text{(simplify)} \\ & = & 4\boxed{\text{2T(n/8) + (n/4)}} + 2n & \text{(third expansion)} \\ & = & 8T(n/8) + 3n & \text{(simplify)} \end{array}\right\} \quad (13)$$

This is the expansion step. At this point, we may guess that the ith expansion, the formula is

$$(G)_i : \quad T(n) = 2^i T(n/2^i) + in. \quad (14)$$

To verify our guess, we use natural induction. Note that the formula (14) is true for $i = 1$ (it also holds for $i = 2$ and 3, but this is not logically necessary). We need an induction step: This amounts to expanding the formula once more:

$$\left.\begin{array}{rcll} T(n) & = & 2^i\boxed{2T(n/2^i)} + in & \text{(guessed } i\text{th expansion)} \\ & = & 2^i\boxed{2T(n/2^{i+1}) + n/2^i} + in & (i+1\text{st expansion}) \\ & = & 2^{i+1}T(n/2^{i+1}) + (i+1)n, & \text{(simplify)} \end{array}\right\} \quad (15)$$

and noting that this confirms that the formula holds for $i+1$ (cf. formula $(G)_{i+1}$ in (14)).

Finally, we must choose a value of i at which to stop this expansion. First consider the ideal situation where n is a power of 2 and we choose $i = \lg n$. Then (14) yields $T(n) = 2^i T(n/2^i) + in = nT(1) + (\lg n)n$. Invoking DIC to make $T(1) = 0$, we obtain the solution $T(n) = n \lg n$. This is a beautiful solution, except for one problem: i must be an integer, and it will not work when n is not a power of 2. It makes no sense to pretend that i is a real variable (as we did for n). In general, we may choose an integer close to $\lg n$: $\lceil \lg n \rceil$ or $\lfloor \lg n \rfloor$ will do. Let us choose

$$i = \lfloor \lg n \rfloor \tag{16}$$

as our stopping value. With this choice, we obtain $1 \leq n/2^i < 2$. Under DIC, we can choose the initial condition to be

$$T(n) = 0, \qquad \text{for } n < 2. \tag{17}$$

This yields the *exact* solution that for $n \geq 2$,

$$T(n) = n \lfloor \lg n \rfloor . \tag{18}$$

¶6. Is is really rote? To recap, there are four distinct stages in the rote method:

(E) Expansion steps as in (13). This is the rote part. You can expand as many times as you like until you see the general pattern.

(G) Guessing of a formula for the ith expansion, as in (14). This guess may require some creativity. Indeed, if we had not re-arranged the terms in our example in the suggestive manner, one might not see the pattern readily. So perhaps "rote" is a misnomer.

(V) Verification of the formula as in (15). This step should be mechanical, and amounts to one more expansion step and re-arranging the terms into the desired form. One problem is that students sometimes do not do this step "honestly" (i.e., do the jump to the conclusion which you expect).

Child's dilemma: *I can't spell* banana *because I don't know when to stop!*

(S) Stopping criteria choice as in (16). You need to know when to stop expansion! Note you must choose i to be a natural number. Thus, you cannot pick "$i = \lg n$" in (16), but need something like $i = \lceil \lg n \rceil$ or $i = \lfloor \lg n \rfloor$. According to DIC, you can pick any i large enough that the recursive term $T(k)$ has an argument k that is below some fixed constant (e.g., $k < 1$). Using DIC, you can declare $T(k)$ to be any value you like (usually $T(k) = 0$ is good).

In general, your guess for the i-th expansion is in the form of a summation $\sum_{j=0}^{i-1} f(j)$ for some function f. If you stop at m-th expansion, you are left with the sum $\sum_{i=0}^{m-1} f(j)$. It just happens that for Mergesort, $f(i)$ is identically equal to n, and so the $\sum_{i=0}^{m-1} n$ is just mn ($m = \lfloor \lg n \rfloor$). Unfortunately, in general, you cannot leave the answer as a sum, and you will need some summation techniques. Summation techniques will be taken up in its own section below. In view of this additional feature, the fourth and last stage might be called the Stop-and-Sum stage.

Since the four stages are Expand, Guess, Verify and Stop-and-Sum, we may also refer to the Rote Method as the **EGVS method**. When the method works, it can give you the exact solution. How can this method fail? It is clear that you can always perform expansions, but you may be stuck at the next step. For instance, try to expand the recurrence $T(n) = 2T(\lceil n/2 \rceil) + n$ in an exact form. The only way out is to give up exact solution, and guess reasonable upper and/or lower bounds.

The appearance of the floor function in the solution (18) makes $T(n)$ discontinuous whenever n is a power of 2. We can make the solution continuous if we fully exploit our freedom in specifying

boundary conditions. Let us now assume that $T(n) = n \lg n$ for $1 \leq n < 2$. Then the above proof gives the solution

$$T(n) = n \lg n \tag{19}$$

for $n \geq 1$. This solution is the "ultimate" in simplicity for the recurrence (8). In the exercises, we see more examples of the influence of DIC (17) on our solution.

EXERCISES

Exercise 3.1: No credit work: Rote is discredited word in pedagogy, so we would like a more dignified name for this method. We could call this the "4-Fold Path". Suggest your own name for this method. In a humorous vein, what could EGVS stand for? ♢

Pronounce "EGVS" as "egg-us" (like the Romans, treat V as U).

Exercise 3.2: Use the EGVS Method to solve the following recurrences
(a) $T(n) = n + 8T(n/2)$.
(b) $T(n) = n + 16T(n/4)$.
(c) Can you generalize your results in (a) and (b) to recurrences of the form $T(n) = n + aT(n/b)$ when a, b are in some special relation? ♢

Exercise 3.3: Solve the Karatsuba recurrence (11) using the Rote Method. HINT: You may want to look ahead to Section 5 on Geometric series. ♢

Exercise 3.4: Give the exact solution for $T(n) = 2T(n/2) + n$ for $n \geq 1$ under the initial condition $T(n) = 0$ for $n < 1$. ♢

Exercise 3.5: Solve (12) assuming that $d(n) = n^\beta$ for some real β. NOTE: there will be three different cases, depending on the relationships between β, a, b. ♢

Exercise 3.6: Let us consider the following form of DIC, where we assume that

$$C_0 \leq T(n) \leq C_1$$

for $0 < n \leq n_1$, with the recurrence operative for $n > n_1$. Here, C_0, C_1, n_1 are positive constants. Solve the Mergesort Recurrence under this initial condition, and show how the solution depends on n_1, C_0, C_1. ♢

END EXERCISES

§4. Real Induction

The rote method, when it works, is a very sharp tool in the sense that as it gives us the exact solution to recurrences. Unfortunately, it does not work for many recurrences: while you can always expand, you may not be able to guess the general formula for the i-th expansion. We now introduce a more widely applicable method, based on the idea of "real induction".

To illustrate this idea, we use a simple example: consider the recurrence

$$T(x) = T(x/2) + T(x/3) + x. \tag{20}$$

The student is encouraged to attempt the rote method on this recurrence. Let us use real induction to prove an upper bound: suppose we guess that $T(x) \leq Kx$ (ev.), for some $K > 1$. Then we verify it "inductively": Try rote first!

$$\begin{array}{rcll} T(x) & = & T(x/2) + T(x/3) + x & \text{(By definition)} \\ & \leq & K\frac{x}{2} + K\frac{x}{3} + x & \text{(Inductive hypothesis)} \\ & = & Kx\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{K}\right) & \\ & \leq & Kx & \text{(Provided } K \geq 6\text{)} \end{array}$$

In the following, we will rigorously justify this method of proof.

How did we guess the upper bound $T(x) \leq Kx$? What if we had guessed $T(x) \leq Kx^2$? Well, we would have succeeded as well. In other words, this argument confirms a particular guess; it does not tell us anything about the optimality of the guess (in reality, the proof do yields hints on how tight an inequality is). We could likewise use real induction to confirm a guessed lower bound. The combined upper and lower bound can often lead to optimal bounds.

¶7. Natural Induction. Real induction is not a familiar in computing or even mathematics, so let us begin by recalling the related but well-known method of **natural induction**. The latter is a proof method based on induction over natural numbers. In brief, suppose $P(\cdot)$ is a natural number predicate, i.e., for each $n \in \mathbb{N}$, $P(n)$ is a proposition.

For example, $P(n)$ might be "There is a prime number between n and $n + 10$ inclusive". A proposition is either true or false. Thus, we may verify[4] that $P(100)$ is true because 101 is prime, but $P(200)$ is false because 211 is the smallest prime larger than 200. A similar predicate is $P(n) \equiv$ "there is prime between n and $2n - 1''$, called Bertrand's Postulate (1845).

We simply write "$P(n)$" or, for emphasis, "$P(n)$ holds" when we want to assert that "proposition $P(n)$ is true". Natural induction is aimed at proving propositions of the form

$$(\forall n \in \mathbb{N})[P(n) \text{ holds}]. \tag{21}$$

When (21) holds, we say the predicate $P(\cdot)$ is **valid**. For instance, Chebyshev proved in 1850 that Bertrand's Postulate $P(n)$ is valid. A "proof by natural induction" has three steps:
(i) [Natural Basis Step] Show that $P(0)$ holds.
(ii) [Natural Induction Step] Show that if $n \geq 1$ and $P(n-1)$ holds then $P(n)$ holds:

$$(n \geq 1) \wedge\ P(n-1) \Rightarrow P(n). \tag{22}$$

(iii) [Principle of Natural Induction] Invoke the principle of natural induction, which simply says that (i) and (ii) imply the validity of $P(\cdot)$, i.e., (21).

Since step (iii) is independent of the predicate $P(\cdot)$, we only need to show the first two steps. A variation of natural induction is the following: for any natural number predicate $P(\cdot)$, introduce a new predicate (the "star version of P") denoted $P^*(\cdot)$, defined via

$$P^*(n) : (\forall m \in \mathbb{N})[m < n \Rightarrow P(m)]. \tag{23}$$

The "Strong Natural Induction Step" replaces (22) in step (ii) by

$$(n \geq 1) \wedge\ P^*(n) \Rightarrow P(n). \tag{24}$$

[4] The smallest n such that $P(n)$ is false is $n = 114$.

It is easy to see that if we carry out the Natural Basis Step and the Strong Natural Induction Step, we have shown the validity of $P^*(n)$. Moreover, $P^*(\cdot)$ is valid iff $P(\cdot)$ is valid. Hence, a proof of the validity of $P^*(\cdot)$ is called a **strong natural induction proof** of the validity of $P(\cdot)$.

¶8. Real Induction. Now we introduce the real analogue of strong natural induction. Unlike natural induction, real induction is rarely discussed in standard mathematical literature, except possibly as a form of transfinite induction. Nevertheless, this topic holds interest in areas such as program verification [2], timed logic [13], and real computational models [4]. We regard it is an important technique for analysis of algorithms.

Real induction is applicable to **real predicates**, *i.e.*, a predicate $P(\cdot)$ such that for each $x \in \mathbb{R}$, we have a proposition denoted $P(x)$. For example, suppose $T(x)$ is a total complexity function that satisfies the Karatsuba recurrence (11) subject to the initial condition $T(x) = 1$ for $x \leq 10$. Let us define the real predicate

$$P(x) : [x \geq 10 \Rightarrow T(x) \leq x^2]. \tag{25}$$

As in (21), we want to prove the **validity** of the real predicate $P(\cdot)$, i.e.,

$$(\forall x \in \mathbb{R})[P(x) \text{ holds}]. \tag{26}$$

In analogy to (23), we transform $P(\cdot)$ into a "star-version of P", defined as follows:

$$P^*_\delta(x) : (\forall y \in \mathbb{R})[y \leq x - \delta \Rightarrow P(y)] \tag{27}$$

where δ is any positive real number. The predicate $P^*_\delta(x)$ is called the **Real Induction Hypothesis** (RIH). When δ is understood, we may simply write $P^*(x)$ instead of $P^*_\delta(x)$.

THEOREM 1 (Principle of Real Induction). *Let $P(x)$ be a real predicate. Suppose there exist real numbers $\delta > 0$ (gap constant) and x_1 (cutoff constant) such that*

(I) [Real Basis Step] *For all $x < x_1$, $P(x)$ holds.*

(II) [Real Induction Step] *For all $x \geq x_1$, $P^*_\delta(x) \Rightarrow P(x)$.*

Then $P(x)$ is valid: for all $x \in \mathbb{R}$, $P(x)$ holds.

The proof of this principle is left as an exercise. It amounts to a reduction to Natural Induction. The principle behind this reduction is a very intuitive property of real numbers: *Given any $\delta > 0$, for every real number x there is a smallest natural number $n(x)$ such that $x \leq n(x)\delta$.* This is also known as the **Archimedean Property** of the reals. We can divide $\mathbb{R}$ into the set $\{Q(k) : k \in \mathbb{N}\}$ of intervals where each interval $Q(i)$ comprises all those x with $n(x) = k$. This is illustrated in Figure 3. We can then prove that the Principle of Real Induction holds over each $Q(k)$ for k, using natural induction.

"Give me a lever long enough and I can move the earth" – Archimedes

Let us apply real induction to real recurrences. Note that its application requires the existence of two constants, x_1 and δ, making it somewhat harder to use than natural induction.

¶9. Example. Suppose $T(x)$ satisfies the recurrence

$$T(x) = x^5 + T(x/a) + T(x/b) \tag{28}$$

where $a \geq b > 1$. Given $x_0 \geq 1$ and $K > 0$, let $P(x)$ be the proposition

$$x \geq x_0 \Rightarrow T(x) \leq Kx^5. \tag{29}$$

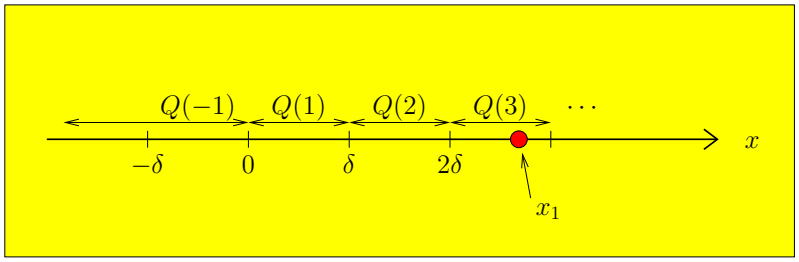


Figure 3: Discrete steps in real induction

Define the constant $k_0 = a^{-5} + b^{-5}$. CLAIM: If $k_0 < 1$ then for all $x_0 \geq 1$, there is a $K > 0$ such that $P(x)$ is valid.

Proof: Now for any x_1, if $x_1 > x_0$ then our Default Initial Condition says that there is a $C > 0$ such that

$$T(x) \leq C$$

for all $x_0 \leq x < x_1$. If we choose K such that $K \geq C/x_0^5$ then for all $x_0 \leq x < x_1$, we have $T(x) \leq C \leq Kx_0^5 \leq Kx^5$ (since $x \geq x_0 \geq 1$). Hence $P(x)$ holds. This establishes the Real Basis Step (I) for $P(x)$ relative to x_1.

To establish the Real Induction Step (II), we need more properties for x_1 and must choose a suitable δ. First choose

$$x_1 = ax_0. \tag{30}$$

Thus for $x \geq x_1$, we have $x_0 \leq x/a \leq x/b$. Next choose

$$\delta = x_1 - (x_1/b) = x_1 \frac{b-1}{b}. \tag{31}$$

This ensures that for $x \geq x_1$, we have $x/a \leq x/b \leq x - \delta$. The Real Induction Hypothesis $P^*_\delta(x)$ says that for all $y \leq x - \delta$, $P(y)$ holds, i.e., $y \geq x_0 \Rightarrow P(y)$. Suppose $x \geq x_1$ and $P^*_\delta(x)$ holds. We need to show that $P(x)$ holds:

$$\begin{aligned} T(x) &= x^5 + T(x/a) + T(x/b) \\ &\leq x^5 + K \cdot (x/a)^5 + K \cdot (x/b)^5, \qquad (\text{by } P^*_\delta(x) \text{ and } x_0 \leq x/a \leq x/b \leq x - \delta) \quad (32) \\ &= x^5(1 + K \cdot k_0) \\ &\leq Kx^5 \qquad (33) \end{aligned}$$

where the last inequality is true provided our choice of K above further satisfies $1 + K \cdot k_0 \leq K$ or $K \geq 1/(1 - k_0)$. This proves the Real Induction Step (II). Invoking the Principle of Real Induction, we conclude that $P(\cdot)$ is valid.

∎

In a similar vein, we can use real induction to prove a lower bound: there is a constant $k > 0$ such that $T(x) \geq kx^5$ (ev.). Hence, we have shown $T(x) = \Theta(n^5)$ for the recurrence (28).

¶10. Default Real Basis. The last example shows that the direct application of the Principle of Real Induction can be tedious, as we have to track constants such as δ, x_1 and K. But this tedium is only associated with justifying the Real Basis (RB); the proof of the Real Induction (RI) is actually not

tedious and highly instructive. Our goal is this subsection is to seek ways to avoid RB, so that you can focus on the interesting part (RI).

There is a simple way out, by fiat! Let $f(x)$ be a complexity function and T satisfies some recurrence. Suppose we want to show that

$$T(x) \prec f(x)$$

by real induction. This amounts to showing that there exists $K > 0$ and x_1 such that

$$(\forall x \geq x_1) T(x) \leq K f(x). \tag{34}$$

We ask you to assume (34) holds provided K and x_1 is sufficiently large. Call this the **Default Real Basis** (DRB). In the next subsection, we will formally justify this for a large class of situations (surely enough to cover all your applications).

¶11. Growth Functions. We will now show that under some general conditions, the Real Basis (RB) of Real Induction Principle is automatic. The idea is to exploit the following property that most natural complexity functions satisfy. Skip on first reading!

A real function $f : \mathbb{R}^k \to \mathbb{R}$ is said to be a **growth function** if f is eventually defined, eventually non-decreasing and is unbounded in each of its variables. For instance, $f(x) = x^2 - 3x$ and $f(x, y) = x^y + x/\log x$ are growth functions, but $f(x) = -x$ and $f(x, y, z) = xy/z$ are not.

THEOREM 2. *Assume $T(x)$ satisfies the real recurrence*

$$T(x) = G(x, T(g_1(x)), \ldots, T(g_k(x)))$$

and

- *$G(x, t_1, \ldots, t_k)$ and each $g_i(x)$ ($i = 1, \ldots, k$) are growth functions.*
- *There is a constant $\delta > 0$ such that each $g_i(x) \leq x - \delta$* (ev. x).

Suppose $f(x)$ is a growth function such that

$$G(x, Kf(g_1(x)), \ldots, Kf(g_k(x))) \leq Kf(x)) \text{ (ev. } K, x). \tag{35}$$

Under the Default Initial Condition, we conclude

$$T(x) = \mathcal{O}(f(x)).$$

Proof. Pick $x_0 > 0$ and $K > 0$ large enough so that all the "eventual premises" of the theorem are satisfied. In particular, $f(x), G(x, t_1, \ldots, t_k)$ and $g_i(x)$ are all defined, non-decreasing and positive when their arguments are $\geq x_0$. Also, $g_i(x_0) \leq x_0 - \delta$ for each i. Let $P(x)$ be the predicate

$$P(x) : \; x \geq x_0 \Rightarrow T(x) \leq Kf(x).$$

Pick

$$x_1 = \max\{g_i^{-1}(x_0) : i = 1, \ldots, k\}. \tag{36}$$

The inverse g_i^{-1} of g_i is undefined at x_0 if there does not exist y_i such that $g_i(y_i) = x_0$, or if there exists more than one such y_i. In this case, take $g_i^{-1}(x_0)$ in (36) to be any y_i such that $g_i(y_i) \geq x_0$. We then conclude that for all $x \geq x_1$,

$$x_0 \leq g_i(x) \leq x - \delta.$$

By the Default Initial Condition (DIC), we conclude that for all $x \in [x_0, x_1]$, $P(x)$ holds. Thus, the Real Basis Step is verified. We now verify the Real Induction Step. Assume $x \geq x_1$ and $P^*_\delta(x)$. Then,

$$\begin{array}{rcll} T(x) & = & G(x, T(g_1(x)), \ldots, T(g_k(x))) & \\ & \leq & G(x, Kf(g_1(x), \ldots, Kf(g_1(x))) & (\text{by } P^*_\delta(x)) \\ & \leq & Kf(x) & (\text{by (35)}). \end{array}$$

Thus $P(x)$ holds. By the Principle of Real Induction, $P(x)$ is valid. This implies $T(x)\mathcal{O}(f(x))$. **Q.E.D.**

To apply this theorem, the main property to verify is the inequality (35), since the other properties are usually routine to check. Let us see this in action on the example (28). We basically need to verify that

1. $f(x) = x^5$, $G(x, t_1, t_2) = x^5 + t_1 + t_2$, $g_1(x) = x/a$ and $g_2(x) = x/b$ are growth functions
2. $g_1(x) \leq x - 1$ and $g_2(x) \leq x - 1$ when x is large enough.
3. The inequality (35) holds when $K \geq 1/(1 - k_0)$. This is just the derivation of (33) from (32).

From theorem 2 we conclude that $T(x) = \mathcal{O}(f(x))$. The step (35) is the most interesting step of this derivation.

It is clear that we can give an analogous theorem which can be used to easily establish lower bounds on $T(x)$. We leave this as an Exercise.

- One phenomenon that arises is that one often has to introduce a stronger induction hypothesis than the actual result aimed for. For instance, to prove that $T(x) = \mathcal{O}(x \log x)$, we may need to guess that $T(x) = Cx \log x + Dx$ for some $C, D > 0$. See the Exercises below.
- A real predicate P can be identified with a subset S_P of $\mathbb{R}$ comprising those x such that $P(x)$ holds. The statement $P(x)$ can be generically viewed as asserting membership of x in S_P, *viz.*, "$x \in S_P$". Then a principle of real induction is just one that gives necessary conditions for a set S_P to be equal to $\mathbb{R}$. Similarly, a natural number predicate is just a subset of $\mathbb{N}$.

In the rest of this chapter, we indicate other systematic pathways; similar ideas are in lecture notes of Mishra and Siegel [14], the books of Knuth [11], Greene and Knuth [8]. See also Purdom and Brown [16] and the survey of Lueker [12].

EXERCISES

Exercise 4.1: Prove theorem 1, by reduction to natural induction. You can also use a proof by contradiction. ♢

Exercise 4.2: Suppose $T(x) = 5T(x/2) + x$. Show by real induction that $T(x) = \Theta(x^{\lg 5})$. ♢

Exercise 4.3: Similar to previous problem, but consider the recurrence $T(x) = 5T(x/2) + x^2$. ♢

Exercise 4.4: Show by real induction that $T(x) = 9T(x/2) + x^3$ that $T(x) \leq K9^{\lg x} - K'x^3$. What is the smallest value of K' you can use? ♢

Exercise 4.5: Consider equation (8), $T(n) = 2T(n/2) + n$. Fix any $k > 1$. Show by induction that $T(n) = \mathcal{O}(n^k)$. Which part of your argument suggests to you that this solution is not tight? ♢

Exercise 4.6: Consider the recurrence $T(n) = n + 10T(n/3)$. Suppose we want to show $T(n) = \mathcal{O}(n^3)$.
(a) Give a proof by real induction.
(b) Suppose $T(n) = n + 10T((n + K)/2)$ for some constant K. How does your proof in (b) change? ♢

Exercise 4.7: Let $T(n) = 2T(\frac{n}{2} + c) + n$ for some $c > 0$.
(a) By choosing suitable initial conditions, prove the following bounds on $T(n)$ by induction, and *not* by any other method:
(a.1) $T(n) \leq D(n - 2c)\lg(n - 2c)$ for some $D > 1$. Is there a smallest D that depends only on c? Explain. Similarly, show $T(n) \geq D'(n - 2c)\lg(n - 2c)$ for some $D' > 0$.
(a.2) $T(n) = n \lg n - o(n)$.
(a.3) $T(n) = n \lg n + \Theta(n)$.
(b) Obtain the exact solution to $T(n)$.
(c) Use your solution to (b) to explain your answers to (a). ♢

Exercise 4.8: Generalize our principle of real induction so that the constant δ is replaced by a real function $\delta : \mathbb{R} \to \mathbb{R}_{>0}$. ♢

Exercise 4.9: (Gilles Dowek, "Preliminary Investigations on Induction over Real Numbers", manuscript 2002).
(a) A set $S \subseteq \mathbb{R}$ is closed if every limit point of S belongs to S. Let $P(x)$ be a real predicate $P(x)$. Assume $\{x \in \mathbb{R} : P(x)\text{holds}\}$ is a closed set. Suppose

$$P(a). \wedge .(\forall c \geq a)[P(c). \Rightarrow .(\exists \varepsilon)(\forall y)[c \leq y \leq c + \varepsilon \Rightarrow P(y)]]$$

Conclude that $(\forall x \geq a)P(x)$.
(b) Let $a, b \in \mathbb{R}$ and $\alpha, \beta : \mathbb{R} \to \mathbb{R}$ such that for all x, $\alpha(x) \geq 0$ and $\alpha(x) > 0$. Suppose f is a differentiable function satisfying

$$f(a) = bf'(x) = -\alpha(x)f(x) + \beta(x)$$

then for all $x \geq a$, $f(x) > 0$. Intuition: If $f(x)$ is the height of an object at time x, then the object will never reach the ground, *i.e.*, $f(x) > 0$. ♢

END EXERCISES

§5. Basic Sums

In this section, we discuss some well-known basic sums and their role in solving recurrences.

¶12. Rote expansion of the Master Recurrence. As motivation, let us return to the rote or EGVS method. We have used it for the Mergesort recurrence (8). We now try apply the technique to the more general Master Recurrence (12) which is

$$T(n) = aT(n/b) + d(n)$$

for $a > 0$ and $b > 1$. Expanding, guessing and verifying yields:

$$\begin{aligned} T(n) &= a\boxed{\text{T(n/b)}} + d(n) \\ &= a^2\boxed{T(n/b^2)} + ad(n/b) + f(n) \\ &= \cdots \\ &= a^i\boxed{T(n/b^i)} + \sum_{j=0}^{i-1} a^j d(n/b^j). \end{aligned}$$

Let us stop when $i = \lfloor \log_b n \rfloor$. Then $n/b^i < b$. We may assume DIC with $T(n) = 0$ for $n < b$. This gives us

$$T(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j d(n/b^j). \tag{37}$$

This solution, unlike in the Mergesort case, is an **open sum**, i.e., a sum with an unbounded number of summands depending on n. We do not regard an open sum as a satisfactory solution. Thus the last step in the EGVS method is stop-and-sum. This summing part is the topic of this section.

¶13. The Standard Recurrence and Descending Sums. Basically, the EGVS method has transformed the Master Recurrence into a recurrence of the form

$$T(n) = T(n-1) + f(n). \tag{38}$$

We shall call this the **standard recurrence**. Our goal in the following sections is to show systematic ways to reduce many recurrences into this standard form. Trivially, (38) has the following open sum as solution

$$T(n) = \sum_{i=1}^{n} f(i), \tag{39}$$

assuming $T(0) = 0$ and n is integer.

In the solution (39) we have assumed that n is integer. But what if n is an arbitrary real value? Let us introduce some general notations that befits our intention of "going totally real". In general, for any real numbers a, b, we define two kinds of sums of f-values over this real interval $[a, b]$:

So $\sum_{x\geq 1}^{\pi} x = 3\pi - 3$ where $\pi = 3.1415\ldots$.

$$\left.\begin{array}{rcll} \sum_{i\geq a}^{b} f(i) & = & f(b) + f(b-1) + f(b-2) + \cdots + f(b - \lfloor b-a \rfloor) & (descend) \\ \sum_{i=a}^{b} f(i) & = & f(a) + f(a+1) + f(a+2) + \cdots + f(a + \lfloor b-a \rfloor) & (ascending) \end{array}\right\} \tag{40}$$

We call these the **descending** and **ascending** f**-summations**. Such sums are defined to be 0 if $a > b$. Note that the difference between these two sums is indicated by the way we write the initial value of the summation variable i: "$\sum_{i\geq a}^{b}$" instead "$\sum_{i=a}^{b}$". We shall mainly focus on the descending sums, but sometimes we need to use ascending sums as well. There is a simple connection between the these two sums:

Henceforth, pay close attention to this minute detail!

$$\sum_{i\geq a}^{b} f(i) = \sum_{i=a}^{b} f(b-i). \tag{41}$$

Even when $f(x)$ is a partial function, these sums are well-defined using the convention that undefined summands are replaced by 0. In recognition of our interest in descending sums, we introduce a convenient notation: for any complexity function f, let

$$S_f(n) := \sum_{i \geq 1}^{n} f(i). \tag{42}$$

and thus the solution to our standard recurrence (38) is

$$T(n) = S_f(n). \tag{43}$$

¶14. What Does It Mean to Solve a Recurrence? If the open sum in the RHS of (39) is unsatisfactory, what is satisfactory? Let us get a hint using a simple example. Suppose $f(n) = n$ in (39). Then we know how to convert the open sum into a a **closed sum**:

$$T(n) = \sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} i = \binom{n+1}{2} = \frac{n(n+1)}{2} = \Theta(n^2).$$

Indeed, we would be perfectly happy with the answer "$T(n) = \Theta(n^2)$" even though the answer is really $\binom{n+1}{2}$ — remember that we are generally interested in Θ-order answers in this book. The reason we are happy with the answer $\Theta(n^2)$ is because n^2 is a "familiar function". So this section is about how we can write some "basic sums" in terms of such familiar functions. These sums are the ones you must know. You will not be responsible for summations outside this small repertoire of basic sums.

¶15. Familiar Functions. So we conclude that "solving a recurrence" is relative to the form of solution we allow. This we interpret to mean a finite sum or finite product involving only "familiar" functions. For our purposes, the functions considered familiar include

polynomials $f(n) = n^k$, logarithms $f(n) = \log n$, and exponentials $f(n) = c^n$ $(c > 0)$.

I see! Solving means to relate to known functions

Functions such as factorials $n!$, binomial coefficients $\binom{n}{k}$ and harmonic numbers H_n (see below) are tightly bounded by familiar functions, and are therefore considered familiar. Finally, we have a rule saying that *the sum, product and functional composition of familiar functions are considered familiar*. Thus $\log^k n$, $\log \log n$, $n + 2 \log n$ and $n^n \log n$ are familiar. For instance, let $f(n)$ be the number of ways an integer n can be written as the sum of two integers. Number theorists have shown that $f(n)$ is $(\log n)^{O(\log n)}$, which is familiar by our definition. In addition to the above functions, two very slow growing functions arise naturally in algorithmic analysis. These are the log-star function $\log^* x$ and the inverse Ackermann function $\alpha(n)$ (see Lecture XII). We will consider them familiar, although functional compositions involving such strange functions are only "familiar" in our very technical sense!

We refer the reader to the Appendix A in this lecture for basic properties of the exponential and logarithm function. Here are some simple facts that you should know of some familiar functions:

LEMMA 3.
(i) For all $k < k'$, $n^k = \mathcal{O}(n^{k'})$ and $n^k \neq \Omega(n^{k'})$.
(ii) For all $k > 0$, $\lg n = \mathcal{O}(n^k)$ and $\lg n \neq \Omega(n^k)$.
(iii) For all k and all $c > 1$, $n^k = \mathcal{O}(c^n)$ and $n^k \neq \Omega(c^k)$.

We ask you to prove these in the exercises.

¶16. Arithmetic series. The basic arithmetic series is

$$S_n := \sum_{i=1}^{n} i = \binom{n+1}{2}. \tag{44}$$

In proof,

$$2S_n = \sum_{i=1}^{n} i + \sum_{i=1}^{n}(n+1-i) = \sum_{i=1}^{n}(n+1) = n(n+1).$$

There is a well-known "proof by picture" where you draw two congruent staircases, each representing the desired sum; you can put these two staircases together to get a rectangle of area $2S_n = n(n+1)$.

More generally, for fixed $k \geq 1$, we have the "arithmetic series of order k",

$$S_n^k := \sum_{i=1}^{n} i^k = \Theta(n^{k+1}). \tag{45}$$

In proof, we have

$$n^{k+1} > S_n^k > \sum_{i=\lceil n/2 \rceil}^{n} (n/2)^k \geq (n/2)^{k+1}.$$

For more precise bounds, we bound S_n^k by integrals,

$$\frac{n^{k+1}}{k+1} = \int_0^n x^k dx < S_n^k < \int_1^{n+1} x^k dx = \frac{(n+1)^{k+1}-1}{k+1},$$

yielding

$$S_n^k = \frac{n^{k+1}}{k+1} + \mathcal{O}_k(n^k). \tag{46}$$

Don't worry about the integrals here — we will find alternatives below. But do not knock calculus – it is very useful for some situations, even if not in this book.

¶17. Geometric series. For $x \neq 1$ and $n \geq 1$,

$$\begin{aligned} S_n(x) &:= \sum_{i=0}^{n-1} x^i \\ &= \frac{x^n - 1}{x-1}. \end{aligned} \tag{47}$$

In proof, note that $xS_n(x) - S_n(x) = x^n - 1$. Next, letting $n \to \infty$, we get the series

$$\begin{aligned} S_\infty(x) &:= \sum_{i=0}^{\infty} x^i \\ &= \begin{cases} \infty & \text{if } x \geq 1 \\ \uparrow \text{ (undefined)} & \text{if } x \leq -1 \\ \frac{1}{1-x} & \text{if } |x| < 1. \end{cases} \end{aligned}$$

Why is $S_\infty(-1)$ (say) considered undefined? For instance, writing

$$\begin{aligned} S_\infty(-1) &= 1-1+1-1+1-1+\cdots \\ &= (1-1)+(1-1)+(1-1)+\cdots \\ &= 0+0+0+\cdots, \end{aligned}$$

we conclude $S_\infty(-1) = 0$. But writing

$$\begin{aligned} S_\infty(-1) &= 1-1+1-1+1-\cdots \\ &= 1-(1-1)+(1-1)-\cdots \\ &= 1+0+0+\cdots, \end{aligned}$$

we conclude $S_\infty(-1) = 1$. So that we must consider this sum as having no definite value, i.e., undefined. Again,

$$\begin{aligned} S_\infty(-1) &= 1-1+1-1+1-\cdots \\ &= 1-S_\infty(-1), \end{aligned}$$

Mathematical analysts learned this lesson in the 19th century: treat infinite sums with great care.

and we conclude that $S_\infty(-1) = 1/2$. In fact, $S_\infty(-1)$ can take infinitely many possible values in this way. This provides a strong case why $S_\infty(-1)$ should be regarded as undefined.

Viewing x as a formal[5] variable, the simplest infinite series is $S_\infty(x) = \sum_{i=0}^{\infty} x^i$. It has a very simple closed form solution,

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}. \tag{48}$$

The one infinite series to know!

Viewed numerically, we may regard this solution as a special case of (47) when $n \to \infty$; but avoiding numerical arguments, it can be directly derived from the formal identity $S_\infty(x) = 1 + xS_\infty(x)$. We suggest calling $\sum_{i=0}^{\infty} x^i$ the "mother of series" because, from the formal solution to this series, we can derive solutions for many related series, including finite series. In fact, for $|x| < 1$, we can derive equation (47) by plugging equation (48) into

$$S_n(x) = S_\infty(x) - x^n S_\infty(x) = (1-x^n)S_\infty(x).$$

By differentiating both sides of the mother series with respect to x, we get:

$$\begin{aligned} \frac{1}{(1-x)^2} &= \sum_{i=1}^{\infty} i x^{i-1} \\ \frac{x}{(1-x)^2} &= \sum_{i=1}^{\infty} i x^i \end{aligned} \tag{49}$$

This process can be repeated to yield formulas for $\sum_{i=0}^{\infty} i^k x^i$, for any integer $k \geq 2$. Differentiating both sides of equation (47), we obtain the finite summation analogue:

$$\sum_{i=1}^{n-1} i x^{i-1} = \frac{(n-1)x^n - nx^{n-1} + 1}{(x-1)^2}, \tag{50}$$

$$\sum_{i=1}^{n-1} i x^i = \frac{(n-1)x^{n+1} - nx^n + x}{(x-1)^2}, \tag{51}$$

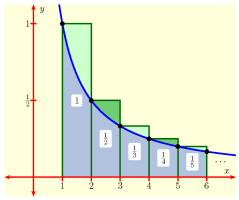
Combining the infinite and finite summation formulas, equations (49) and (50), we also obtain

$$\sum_{i=n}^{\infty} i x^i = \frac{nx^n - (n-1)x^{n+1}}{(1-x)^2}. \tag{52}$$

[5] I.e., as an uninterpreted symbol rather than as a numerical value. Thereby, we avoid questions about the sum converging to some unique numerical value.

We may verify by induction that these formulas actually hold for all $x \neq 1$ when the series are finite. In general, for any $k \geq 0$, we obtain formulas for the **geometric series of order** k:

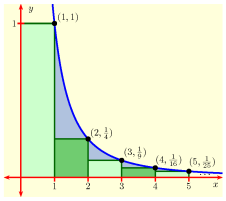
$$\sum_{i=1}^{n-1} i^k x^i. \tag{53}$$



The infinite series have finite values only when $|x| < 1$.

¶18. Harmonic series. For natural numbers $n \geq 1$, the nth **harmonic number** is defined as

$$H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}. \tag{54}$$



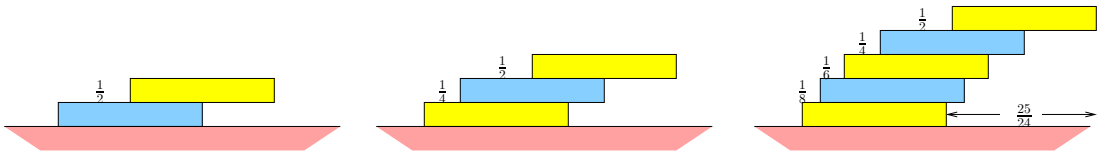
We can give some easy estimates of H_n using using calculus:

$$H_n < 1 + \int_1^n \frac{dx}{x} < 1 + H_n.$$

But $\int_1^n \frac{dx}{x} = \ln n$. This proves that

$$H_n = \ln n + g(n), \qquad \text{where } 0 < g(n) < 1. \tag{55}$$

Note that $\ln$ is the natural logarithm (appendix A).

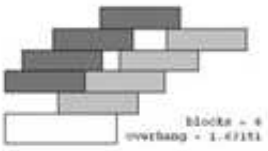


(a) $n = 2$ (b) $n = 3$ (c) $n = 5$

Figure 4: Stacking bricks with maximum overhang: for $n = 5$, overhang is more than one brick length!

Does your architecture friends know about this one?

Harmonic numbers arise naturally in the analysis of algorithms. But here is a "physical" application of harmonic numbers: Suppose you have a set of $n \geq 2$ bricks. The bricks are identical and have unit length. We want to stack the bricks so that the overhang is as large as possible. For instance, if $n = 2$, the overhang is $1/2$ since we can put one brick over the other such that the center of gravity of the top brick is above the edge of the bottom brick. This is illustrated in Figure 4(a). The case of $n = 3$, we may check that the overhang is $3/4$ (Figure 4(b)). An obvious question is whether we can make the overhang arbitrarily large (provided n is large enough)? Somewhat surprisingly, the answer is 'yes'. See Figure 4(c) for the case $n = 5$: in this case, the overhang is $25/24$, already exceeding the length of a single brick! How many bricks do we need to have an overhang exceeding two brick lengths? In general, the overhang is $\frac{1}{2}H_{n-1}$ (Exercise). As H_n is about $\ln n$, the overhang goes to infinity (albeit very slowly) as $n \to \infty$.
For more information, see the fascinating book "How Round is Your Circle? Where Engineering and Mathematics Meet", by John Bryant and Chris Sangwin (Princeton University Press, 2008). This solution is based on an assumption that you stack at most one brick on another. What if you allow more than one? You can do a lot better than the above classical solution! Mike Paterson and Uri Zwick (2009, American Math. Monthly) have investigated the case of multiple stacking. The maximum overhang for 8 bricks are illustrated in the margin here.



We can view (55) as a special case of our descending sums $S_f(n)$ where $f(n) = 1/n$. Then for all real n, $H_n = S_f(n) = \sum_{i \geq 1}^{n} \frac{1}{n}$. Here is a more precise estimate for $g(n)$: for $n \geq 1$,

$$\gamma + \frac{1}{2n} - \frac{1}{8n^2} < g(n) < \gamma + \frac{1}{2n} \tag{56}$$

where $\gamma = 0.577...$ is **Euler's constant**. See Polya and Szego, Problems and Theorems in Analysis, Volume I, Springer-Verlag, Berlin (1972).

We can also deduce asymptotic properties of H_n without calculus: if $n = 2^N$, then

$$H_n = \sum_1 + \sum_2 + \cdots + \sum_N$$

where $\sum_k$ is defined as $\sum_{i=2^{k-1}}^{2^k - 1} \frac{1}{i}$. Since $\sum_k$ has 2^{k-1} terms, and each term is between $1/2^k$ and $1/2^{k-1}$, we obtain

$$1/2 = 2^{k-1}\frac{1}{2^k} < \sum_k \leq 2^{k-1}\frac{1}{2^{k-1}} = 1.$$

This proves that

$$\frac{1}{2}N \leq H_n \leq N$$

for n a power of 2. Extrapolating to all values of n, we conclude that $H_n = \Theta(N) = \Theta(\log n)$. Since we may choose N as big as we like, *we have also shown that H_n and $\lg(n)$ are unbounded*. This proof idea can be extended (see below).

¶19. Stirling's Approximation. So far, we have treated open sums. If we have an open product such as the factorial function $n!$, we can convert it into an open sum by taking logarithms. This method of estimating an open product may not give as tight a bound as we wish (why?). For the factorial function, there is a family of more direct bounds that are collectively called **Stirling's approximation**. The following Stirling approximation is from Robbins (1955) and it may be committed to memory:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\, e^{\alpha_n}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}.$$

Sometimes, the bound $\alpha_n > (12n)^{-1} - (360n^3)^{-1}$ is useful [5]. Up to Θ-order, Stirling's approximation simplifies to

$$n! = \Theta\left(\left(\frac{n}{e}\right)^{n+\frac{1}{2}}\right).$$

¶20. Binomial theorem.

$$\begin{aligned}(1+x)^n &= 1 + nx + \frac{n(n-1)}{2}x^2 + \cdots + x^n \\ &= \sum_{i=0}^{n} \binom{n}{i} x^i.\end{aligned}$$

For solving real recurrences, it is useful to generalize this theorem to $(1+x)^p$ for any real number p. In general, the binomial function $\binom{n}{i}$ may be extended to all real p and integer i as follows:

$$\binom{p}{i} = \begin{cases} 0 & \text{if } i < 0 \\ 1 & \text{if } i = 0 \\ \frac{p(p-1)\cdots(p-i+1)}{i(i-1)\cdots 2\cdots 1} & \text{if } i > 0. \end{cases}$$

We use Taylor's expansion for a function $f(x)$ at $x = a$:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots$$

where $f^{(n)}(x) = \frac{d^n f}{d^n x}$. This expansion is defined provided all derivatives of f exist and the series converges. Applied to $f(x) = (1+x)^p$ for any real p at $x = 0$, we get the desired binomial theorem for real exponents:

$$\begin{aligned} (1+x)^p &= 1 + px + \frac{p(p-1)}{2!}x^2 + \frac{p(p-1)(p-2)}{3!}x^3 + \cdots \\ &= \sum_{i \geq 0} \binom{p}{i} x^i. \end{aligned}$$

See [11, p. 56] for Abel's generalization of the binomial theorem.

EXERCISES

Exercise 5.1: Show Lemma 3. For logarithms, please use direct inequalities (no calculus). ♢

Exercise 5.2: Strengthen the lower bounds in Lemma 3 from $\neq \Omega(f(n))$ to $= o(f(n))$. ♢

Exercise 5.3: Let $h(n)$ denote the maximum overhang for n bricks. Prove that $h(n) = \sum_{i=1}^{n-1} \frac{1}{2i} = \frac{1}{2}H_{n-1}$. Thus, $h(2) = 1/2$, $h(3) = h(2) + 1/4 = 3/4$, $h(4) = h(3) + 1/6 = 11/12$, and $h(5) = h(4) + 1/8 = 25/24$. HINT: Let the right edge of the ith brick be at position x_i where the ith brick is stacked on the $i+1$st brick with $x_i > x_{i+1}$. Inductively, assume that the optimal configuration for $h(n)$ is $(x_1, x_2, \ldots, x_n)$ where $x_i - x_{i+1} = 1/2i$. Moreover, the C.G. of the optimal configuration for $h(n-1)$ is at x_n. Extend this induction hypothesis to $h(n+1)$. ♢

Exercise 5.4: Let $c > 0$ be any real constant.
(a) Show that $\ln(n+c) - \ln n = \mathcal{O}(c/n)$.
(b) Show that $|H_{x+c} - H_x| = \mathcal{O}(c/n)$ where H_x is the generalized Harmonic function.
(c) Bound the sum $\sum_{i=1+\lfloor c \rfloor}^{n} \frac{1}{i(i-c)}$. ♢

Exercise 5.5: Consider $S_\infty(x)$ as a numerical sum.
(a) Prove that there is a unique value for $S_\infty(x)$ when $|x| < 1$.
(b) Prove that there are infinitely many possible values for $S_\infty(x)$ when $x \leq -1$.
(c) Are all real values possible as a solution to $S_\infty(-1)$? ♢

Exercise 5.6: Show the following useful estimate: $\ln(n) - (2/n) < \ln(n-1) < (\ln n) - (1/n)$. ♢

Exercise 5.7:

(a) Give the exact value of $\sum_{i=2}^{n} \frac{1}{i(i-1)}$. HINT: use partial fraction decomposition of $\frac{1}{i(i-1)}$.

(b) Conclude that $H_{\infty}^{(-2)} \leq 2$. ♢

Exercise 5.8: (Basel Problem) The goal is to give tight bounds for $H_n^{(-2)} := \sum_{i=1}^{n} \frac{1}{i^2}$ (cf. previous exercise).

(a) Let $S(n) = \sum_{i=2}^{n} \frac{1}{(i-1)(i+1)}$. Find the exact bound for $S(n)$.

(b) Let $G(n) = S(n) - H_n^{(-2)} + 1$. Now $\gamma' = G(\infty)$ is a real constant,

$$\gamma' = \frac{1}{1\cdot 3\cdot 4} + \frac{1}{2\cdot 4\cdot 9} + \frac{1}{3\cdot 5\cdot 16} + \cdots + \frac{1}{(i-1)\cdot(i+1)\cdot i^2} + \cdots .$$

Show that $G(n) = \gamma' - \theta(n^{-3})$.

(c) Give an approximate expression for $H_n^{(-2)}$ (involving γ') that is accurate to $\mathcal{O}(n^{-3})$. Note that γ' plays a role similar to Euler's constant γ for harmonic numbers.

(d) What can you say about γ', given that $H_{\infty}^{(-2)} = \pi^2/6$? Use a calculator (and a suitable approximation for π) to compute γ' to 6 significant digits. ♢

Exercise 5.9: Solve the recurrence $T(n) = 5T(n-1) + n$. ♢

Exercise 5.10: Solve exactly (choose your own initial conditions):

(a) $T(n) = 1 + \frac{n+1}{n} T(n-1)$.

(b) $T(n) = 1 + \frac{n+2}{n} T(n-1)$. ♢

Exercise 5.11: Show that $\sum_{i=1}^{n} H_i = (n+1)H_n - n$. More generally,

$$\sum_{i=1}^{n} \binom{i}{m} H_i = \binom{n+1}{m+1} \left[H_{n+1} - \frac{1}{m+1} \right].$$

♢

Exercise 5.12: (J.van de Lune, 1980) Above, we defined $H_n := \sum_{i\geq 1}^{n} 1/i$ (descending sum). A variant that is neither a descending nor an ascending sum is to define $H(a,b) := \sum_{a\leq i\leq b} 1/i$ where the summation is over all integer values of i in the range $[a,b]$. Then this sum is bounded by

$$\sum_{a\leq x\leq b} \frac{1}{x} \leq \ln(y/x) + \min\{1, 1/x\}$$

♢

Exercise 5.13: Give a recurrence for S_n^k (see (45)) in terms of S_n^i, for $i < k$. Solve exactly for S_n^4. ♢

Exercise 5.14: Derive the formula for the "geometric series of order 2", $k = 2$ in (53). ♢

Exercise 5.15: (a) Use Stirling's approximation to give an estimate of the exponent E in the expression $2^E = \binom{2n}{n}$.
(b) (Feller) Show $\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k}^2$. ♢

Exercise 5.16: Suppose your architecture friend said that your brick tower design in an above exercise is not realistic (we have to admit this). That design was able to achieve arbitrarily large overhangs based on the fact that the harmonic numbers H_n tending to infinity. Here is a sequence of numbers that tend to infinity but slower: $G_n = \sum_{i=1}^{n} \frac{1}{i \lg i}$. Can you design a similar overhanging tower based on this sequence? Can you convince your architecture friend that this is stable enough to build? ♢

END EXERCISES

§6. Standard Form and Summation Techniques

Recall that our goal is to reduce all recurrences to the **standard form**:

$$t(n) = t(n-1) + f(n). \tag{57}$$

We have noted that the solution is the descending sum

$$t(n) = S_f(n) = \sum_{i \geq 1}^{n} f(i) \tag{58}$$

assume DIC with $t(n) = 0$ for $n < 1$. It is perhaps instructive to see this derived in another stylized way known as "telescopy". Assuming the recurrence is valid for all $n \geq 1$, we have

$$t(n-i+1) - t(n-i) = f(n-i+1), \quad (i = 1, \ldots, \lfloor n \rfloor).$$

Adding these $\lfloor n \rfloor$ equations together, all but two terms on the left-hand side cancel, leaving us

$$t(n) - t(n - \lfloor n-1 \rfloor) = \sum_{i \geq 1}^{n} f(i).$$

(We say the left-hand side is a "telescoping sum".)

¶21. Polynomial-type and Exponential-type Sums. Let us consider what is to be done if the open sum (58) does not readily reduce to one of the basic sums we have discussed in the previous section. Traditionally, the sum $S_f(n)$ (for $n \in \mathbb{N}$) is solved using the Euler-Maclaurin summation formula. The formula is for ascending sums:

$$\sum_{i=1}^{n-1} f(i) = \int_{i}^{n} f(x)dx + \left(\sum_{i=1}^{\infty} \frac{B_i f^{(i-1)}(x)}{i!} \right)_{x=1}^{x=n}$$

where B_i is the ith Bernoulli number. See [7, p. 217]. But in this book, we emphasize the solution of recurrences using purely elementary arguments, preferring to avoid calculus. This is possible because we seek only Θ-order solutions. We now introduce two elementary summation techniques for this purpose. They are based on the following "growth classification" of real functions: it is assumed that the function f in these definitions satisfy $f \geq 0$ and $f > 0$ (ev.).

No calculus please, we are computer scientists!

Polynomial Type: A real function f is **polynomial-type** if f is non-decreasing (ev.) and there is some $C > 0$ such that

$$f(x) \leq C \cdot f(x/2) \text{ (ev.)}.$$

E.g.,

$$f_0(x) = x, \quad f_1(x) = \log x, \quad f_2(x) = f_0(x)f_1(x), \quad f_3(x) = (f_0(x))^a \ (a > 0).$$

Exponential Type: The function f is **exponential-type** if it increases exponentially or it decreases exponentially:
(a) f **increases exponentially** if there exists real numbers $C > 1$ and $k > 0$ such that

$$f(x) \geq C \cdot f(x-k) \text{ (ev.)}.$$

E.g.,

$$g_0(x) = 2^x, \quad g_1(x) = x!, \quad g_2(x) = g_0(x)g_1(x), \quad g_3(x) = 2^{g_0(x)}$$

(b) f **decreases exponentially** if there exists real numbers $0 < c < 1$ and $k > 0$ such that

$$c \cdot f(x) \leq f(x-k) \text{ (ev.)}.$$

E.g.,

$$h_0(x) = 2^{-x}, \quad h_1(x) = x^{-x}, \quad h_2(x) = h_0(x)h_1(x), \quad h_3(x) = 2^{h_0(x)}$$

In proofs, we usually assume $k = 1$ in the above definition of exponential-types: i.e., if $g(n)$ is increasing exponentially, $g(n) \geq Cg(n-1)$ and if $h(n)$ is decreasing exponentially, $h(n) \leq ch(n-1)$. It should be clear that these arguments generalizes to the more general $k > 0$.

We say that the descending sum $S(n) = S_f(n) := \sum_{x \geq 1}^{n} f(x)$ is polynomial-type or exponential-type, depending on the above classification of f. The following theorem gives a simple rule for bounding such sums.

THEOREM 4 (Summation Rules). *Consider the sum $S_f(n)$:*
(i) If f is polynomial-type, $S(n) = \Theta(nf(n))$.
(ii) If f is exponential-type,

$$S_f(n) = \begin{cases} \Theta(f(n)) & \text{if } f \text{ is increasing exponentially,} \\ \Theta(1) & \text{if } f \text{ is decreasing exponentially.} \end{cases}$$

Proof. (i) For a polynomial-type sum, using the fact that f is non-decreasing, we get the upper bound $S_f(n) \leq \sum_{x=1}^{n} f(n) = nf(n)$. For lower bound, we also need that $f(x) \leq Cf(x/2)$ (ev.) for some $C > 0$:

$$\begin{aligned} S_f(n) &\geq \sum_{x \geq n/2}^{n} f(x) \\ &\geq \sum_{x \geq n/2}^{n} f(n/2) \geq \lfloor n/2 \rfloor f(n/2) \\ &\geq \lfloor n/2 \rfloor \frac{f(n)}{C} = \Omega(nf(n)). \end{aligned}$$

(ii-a) For an increasing exponential sum, there is some $C > 1$, $k > 0$ and $m > 0$ such that for all $n \geq m+1$, we have $f(n) \geq Cf(n-k)$. By increasing k, m, n if necessary, we may assume wlog that $k, n-m \in \mathbb{N}$ and $n \geq m+1$. We can even assume $n-m$ is divisible by k. Thus,

$$S_f(n) = f(n) + f(n-1) + \cdots + f(m+1) + S_f(m)$$

We can subdivide the sum $f(n)+f(n-1)+f(n-2)+\cdots+f(m+1)$ into k different subsums, each of the form

$$f(n-\kappa)+f(n-\kappa-k)+f(n-\kappa-2k)+\cdots+f(m-\kappa+k) \tag{59}$$

for each $\kappa = 0, 1, \ldots, k-1$. The lemma then follows since each subsum satisfies

$$\begin{aligned} & f(n-\kappa)+f(n-\kappa-k)+f(n-\kappa-2k)+\cdots+f(m-\kappa+k) \\ < \quad & f(n-\kappa)\left[1+\frac{1}{C}+\frac{1}{C^2}+\cdots\right] \\ = \quad & f(n-\kappa)\frac{C}{C-1} \\ = \quad & \mathcal{O}(f(n-\kappa)) = \mathcal{O}(f(n)). \end{aligned}$$

(ii-b) For a decreasing exponential sum, there is some $c < 1$, $k > 0$ and $m > 0$ such that for all $n \geq m+1$, we have $f(n+k) \leq cf(n)$ and $f(m) \geq f(n)$. Again wlog, $k, n-m \in \mathbb{N}$ and k divides $n-m$. Then

$$S_f(n) = S_f(m) + f(m+1) + f(m+2) + \cdots f(n)$$

where $f(m+1)+f(m+2)+\cdots+f(n)$ can be broken up into k subsums, as in (59). The result follows since each subsum is bounded as

$$\begin{aligned} & f(m-\kappa+k)+f(m-\kappa+2k)+f(m-\kappa+3k)+\cdots+f(n-\kappa) \\ < \quad & f(m-\kappa+k)\left[1+c+c^2+\cdots\right] \\ \leq \quad & f(m)\frac{1}{1-c}. \end{aligned}$$

Q.E.D.

This theorem say that, to upper bound a polynomial-type sum $S_f(n)$, we can replace *each term* in the sum by its largest term $f(n)$. Similarly, to upper bound an exponential-type sum $S_f(n)$, replace *the entire sum* by its largest term. Thus the theorem shifts the burden of estimating sums to the simpler task of identifying the growth-type of the function f.

Let us illustrate applications of this theorem:

- Polynomial Sums.

$$\sum_{i\geq 1}^{n} i\log i = \Theta(n^2\log n), \qquad \sum_{i\geq 1}^{n} \log i = \Theta(n\log n) \qquad \sum_{i\geq 1}^{n} i^a = \Theta(n^{a+1}) \text{ (where } a > 0). \tag{60}$$

- Exponentially Increasing Sums.

$$\sum_{i\geq 1}^{n} 2^i = \Theta(2^n), \qquad \sum_{i\geq 1}^{n} i^{-5}2^{2^i} = \Theta(n^{-5}2^{2^n}), \qquad \sum_{i\geq 1}^{n} i! = \Theta(n!) \quad . \tag{61}$$

- Exponentially Decreasing Sums.

$$\sum_{i\geq 1}^{n} 2^{-i} = \Theta(1), \qquad \sum_{i\geq 1}^{n} i^2 i^{-i} = \Theta(1), \qquad \sum_{i\geq 1}^{n} i^{-i} = \Theta(1) \quad . \tag{62}$$

Summation that does not fit the framework of Theorem 4 can sometimes be reduced to one that does. A trivial case is where summation we are interested in does not begin with $i = 1$. As another example, consider

$$S := \sum_{i \geq 1}^{n} \frac{i!}{\lg^i n}, \tag{63}$$

which has terms depending on i as well as on the limit n. Write $S = \sum_{i \geq 1}^{n} f(i, n)$ where

$$f(i, n) = \frac{i!}{\lg^i n}.$$

We note that $f(i, n)$ is increasing exponentially for $i \geq 2 \lg n$ (ev. n), since $f(i, n) = \frac{i}{\lg n} f(i-1, n) \geq 2f(i-1, n)$. Hence we may split the summation into two parts, $S = A + B$ where A comprise the terms for which $i \geq 2 \lg n$ and B comprising the rest. Since B is an exponential sum, we have $B = \Theta(f(n, n))$. We can easily use Stirling's estimate for A to see that $A = \mathcal{O}(\log^{3/2} n) = \mathcal{O}(f(n, n))$. Thus $S = \Theta(f(n, n))$.

¶22. A Counter Example. Most common functions we encounter will be either polynomial-type or exponential-type. But the function $f(n) = n^{\ln n}$ is neither. Showing that $f(n)$ is not polynomial-type is easy. The box below proves it is not exponential-type. How do we estimate the sum $S_f(n) := \sum_{x \geq 0}^{n} f(x)$ without the benefit of Theorem 4? In this case, techniques similar to polynomial and exponential sums still give reasonably tight bounds (but not Θ-order): $f(n) \leq S(n) \leq nf(n) \leq f(n)^{1+\varepsilon}$ for any $\varepsilon > 0$.

CLAIM: $f(n) = n^{\ln n}$ is not exponential-type. By way of contradiction, suppose there exists $C_0 > 1$ such that

$$f(n) \geq C_0 f(n-1) \text{ (ev.)}. \tag{64}$$

A well-known bound (see Appendix A) says that for $|x| < 1$,

$$\ln(1+x) < x. \tag{65}$$

Also from (55) and (56), we conclude that

$$\ln n + \gamma \leq H_n \leq \ln n + \gamma + (1/n) \quad \text{(ev.)}. \tag{66}$$

All the following inequalities are to hold eventually:

$$\begin{aligned} \ln n &\leq H_n - \gamma \\ &\leq (1/n) + \ln(n-1) + (1/n) \\ &= \ln(n-1) + (2/n), \end{aligned} \tag{67}$$

We now get a contradiction:

$$\begin{aligned} f(n) &= \left[(n-1)(1+\tfrac{1}{n-1})\right]^{\ln n} \\ &\leq (n-1)^{\ln(n-1)+(2/n)}(1+\tfrac{1}{n-1})^{\ln n} && \text{(by (67))} \\ &= f(n-1)\cdot(n-1)^{2/n}\cdot 2^{\ln(1+\frac{1}{n-1})\ln n} \\ &\leq f(n-1)\cdot 2^{2\ln(n-1)/n}\cdot 2^{\frac{\ln n}{n-1}} && \text{(by (65))} \\ &= f(n-1)\cdot C_1(n) \end{aligned}$$

where $C_1(n) := 2^{2\ln(n-1)/n}\cdot 2^{\frac{\ln n}{n-1}}$. Since $\ln C_1(n) = (2\ln(n-1)/n) + (\ln n/(n-1)) \to 0$ as $n \to \infty$, we conclude that $C_1(n) \leq C_0$ (ev.). This show $f(n) \leq f(n-1)C_0$ (ev.), contradicting (64).

To apply the summation rules Theorem 4, we want to rapidly classify functions according to their growth types. For this purpose, the next lemma showing that these growth types are closed under various operations is helpful:

LEMMA 5. *Let $a \in \mathbb{R}$.*
(a) Polynomial-type functions are closed under addition, multiplication, and raising to any positive power $a > 0$.
(b) Exponential-type functions f are closed under addition, multiplication and raising to any power a. In case $a > 0$, the function f^a will not change its subtype (increasing or decreasing). In case $a < 0$, the function f^a will change its subtype.
(c) If f is polynomial-type and $f > 1$ (ev.) then $\lg f$ is also polynomial-type. If f is exponential-type and $a > 1$ then so is a^f.

Proof. All the inequalities in the following proofs are assumed to hold eventually:
(a) Assume $f(n) \leq Cf(n/2)$ and $g(n) \leq Cg(n/2)$ for some $C > 1$. Then $f(n)+g(n) \leq C(f(n/2)+g(n/2))$, $f(n)g(n) \leq C^2 f(n/2)g(n/2)$, and for any $e > 0$, $f(n)^e \leq C^e f(n/2)^e$.
(b) Assume $g_i(n) \geq Cg_i(n-1)$ and $h_i(n) \leq ch_i(n-1)$. for some $C > 1, c < 1$, and for $i = 0, 1$. Also, let $g = g_0$, $h = h_0$. Closure under addition: $g_0(n) + g_1(n) \geq C(g_0(n-1) + g_1(n-1))$ and $h_0(n)+h_1(n) \leq c(h_0(n-1)+h_1(n-1))$. Closure under product: $g_0(n)g_1(n) \geq C^2 g_0(n-1)g_1(n-1))$ and $h_0(n)h_1(n) \leq c^2 h_0(n-1)h_1(n-1)$. Closure under raising to power e: If $e > 0$, then $g^e(n) \geq C^e g^e(n-1)$ and $h^e(n) \leq c^e h^e(n-1)$ where $C^e > 1$ and $c^e < 1$. If $e < 0$, then $g^e(n) \leq C^e g^e(n-1)$ and $h^e(n) \geq c^e h^e(n-1)$ where $C^e < 1$ and $c^e > 1$.

(c) If f is polynomial-type, then $\log(f(n)) \leq (\log C) + \log(f(n/2)) \leq (1 + (\log C)/c)\log(f(n/2))$, where $\log(f(n/2)) \geq c > 0$ for some constant c. This proves $\log f$ to be polynomial-type.

If g, h is exponential type as in (b), then note that $Cg(n) \geq (C-1) + g(n)$ since $g(n) \geq 1$. Thus

$$\begin{aligned} b^{g(n)} &\geq b^{Cg(n-1)} \geq b^{(C-1)+f(n-1)} \\ &\geq b^{C-1}2^{f(n-1)}. \end{aligned}$$

Q.E.D.

¶23. Generalized Harmonic Numbers. For any real α, it is useful to define the generalized Harmonic number

$$H_n^{(\alpha)} := \sum_{i \geq 1}^{n} i^\alpha$$

using a descending sum. Thus $H_n^{(-1)}$ is just the Harmonic numbers H_n when n is integer. But now, we allow n to be any real number. Also when $\alpha \in \mathbb{N}$, $H_n^{(\alpha)}$ is just the arithmetic series in ¶16. When $\alpha \leq -1$, the sum $H_n^{(\alpha)}$ is bounded as $n \to \infty$; the limiting value $H_\infty^{(\alpha)}$ is the value of the Riemann zeta function at $-\alpha$: $\zeta(\alpha) := \sum_{i=1}^{\infty} n^{-\alpha} = H_\infty^{(-\alpha)}$. For instance, $\zeta(2) = H_\infty^{(-2)} = \pi^2/6$. An exercise below estimates the sum $H_n^{(2)}$: we see that a constant analogous to Euler's γ arises.

Let us determine the Θ-order of $H_n^{(\alpha)}$. For all $n, \alpha \in \mathbb{R}$, define the **generalized Harmonic number**

$$\begin{aligned} H^\alpha(n) &:= n^\alpha + (n-1)^\alpha + (n-2)^\alpha + \cdots + (\{n\}+1)^\alpha \\ &= \sum_{x \geq 1}^{n} x^\alpha, \end{aligned} \tag{68}$$

using the descending sum notation (40). The original harmonic numbers in this notation becomes

$$H_n^{(\alpha)} = H^{-\alpha}(n).$$

"Claim": $\ln x$ is identically 1: $\frac{d(\ln x)}{dx} = \frac{1}{x}$ and $\frac{d(1)}{dx} = \frac{d(x^0)}{dx} = x^{0-1} = \frac{1}{x}$. So $\ln x = 1$.

Also, $H^\alpha(n) = 0$ for $n < 1$.

THEOREM 6. *For all* $\alpha \in \mathbb{R}$,

$$H^\alpha(n) = \Theta \begin{cases} 1 & \text{if } \alpha < -1 \\ \lg n & \text{if } \alpha = -1 \\ n^{\alpha+1} & \text{if } \alpha > -1 \end{cases}$$

Proof. It is best to initially assume $n+1$ is a power of 2. Then

$$\begin{aligned} H^\alpha(n) &= \sum_{k=1}^{\lg(n+1)} \left(\sum_{i=2^{k-1}}^{2^k-1} i^\alpha \right) \\ &= \sum_{k=1}^{\lg(n+1)} 2^k \cdot \Theta\left(2^{k\alpha}\right) \\ &= \sum_{k=1}^{\lg(n+1)} \Theta\left(2^{k(1+\alpha)}\right). \end{aligned}$$

Note that the slick use of Θ in this derivation is capturing upper and lower bounds simultaneously. If explicitly spelled out, you would need to consider the cases $\alpha \geq 0$ and $\alpha < 0$ separately. Now we notice that if $1 + \alpha = 0$ then the sum

Exercise: spell it out!

$$\sum_{k=1}^{\lg(n+1)} \Theta\left(2^{k(1+\alpha)}\right) = \Theta(\lg(n+1)).$$

If $1 + \alpha < 0$, then the sum is decreasing exponentially and Theorem 4 yields

$$\sum_{k=1}^{\lg(n+1)} = \Theta(1).$$

If $1 + \alpha > 0$, then the sum is increasing exponentially and Theorem 4 yields

$$\sum_{k=1}^{\lg(n+1)} = \Theta\left(2^{\lg(n+1)(1+\alpha)}\right) = \Theta\left(n^{1+\alpha}\right).$$

When $n+1$ is not a power of 2, we can replace n by $\overline{n} = 2^{\lceil \lg(n+1) \rceil} - 1$ and $\underline{n} = 2^{\lfloor \lg(n+1) \rfloor} - 1$ for upper and lower bounds (Exercise). **Q.E.D.**

Up to Θ-order, the result unifies the standard bounds for (a) the arithmetic series (45), (b) harmonic numbers (55), and (c) geometric sums (47). Our proof is completely elementary; its basic method of splitting up the sum into a "geometric sequence" of groups is applicable to many other estimates involving logarithms.

Here is an application of generalized Harmonic numbers: to solve the recurrence $T(n) = 2T(n/2) + (n/\lg n)$, we convert it to the standard form

$$t(N) = t(N-1) + 1/N \tag{69}$$

using the substitution $t(N) = T(2^N)/2^N$, where $N = \lg n$ is a real variable. According to (43), $t(N) = H_N^{(-1)}$. Back solving, the original recurrence has solution $T(n) = nH_{\lg n}^{(-1)} = \Theta(n \ln \lg n)$.

¶24. Grouping: Breaking Up into Big and Small Parts. The above example (63) illustrates the technique of breaking up a sum into two parts, one containing the "small terms" and the other containing the "big terms". This is motivated by the wish to apply different summation techniques for the 2 parts, and this in turn determines the cutoff point between small and big terms. Suppose we want to show

$$H_n = \sum_{i \geq 1}^{n} \frac{1}{i} = O(\sqrt{n}).$$

Break H_n into two summations, $H_n = A_n + B_n$ where

$$A_n = \sum_{i \geq 1}^{n - \lfloor n - \sqrt{n} \rfloor} \frac{1}{i}$$

comprises the "big terms" (there are at most $\sqrt{n}$ terms in A_n), and B_n contains the remaining $\lfloor n - \sqrt{n} \rfloor$ "small terms". Then

$$A_n \leq \sum_{i \geq 1}^{n - \lfloor n - \sqrt{n} \rfloor} \frac{1}{i} \leq \sqrt{n}$$

and

$$B_n = \sum_{i \geq n - \lfloor n - \sqrt{n} \rfloor}^{n} \frac{1}{i} \leq \sum_{i=1}^{n} \frac{1}{\sqrt{n}} = \sqrt{n}.$$

Thus $S_n \leq 2\sqrt{n} = O(\sqrt{n})$ as desired.

We can generalize the grouping idea to prove the following:

$$H_n < k n^{1/k} \tag{70}$$

for any integer $k \geq 2$. We break the summation H_n into k subsums, $H_n = A_n(1) + A_n(2) + \cdots + A_n(k)$ where $A_n(1)$ comprises the first $\lceil n^{1/k} \rceil$ terms of H_n, $A_n(2)$ comprises the next $\lceil n^{2/k} \rceil - \lceil n^{1/k} \rceil$ terms, etc, where in general, $A_n(j)$ comprises the next $\lceil n^{j/k} \rceil - \lceil n^{(j-1)/k} \rceil$ terms. It is easy to see that each $A_n(j)$ is bounded by $n^{1/k}$ and this proves (70). This proves that H_n is $O(n^c)$ for any $c > 0$. This also implies

$$H_n = o(n^c), \qquad \log_b n = o(n^c).$$

EXERCISES

Exercise 6.1: Verify that the examples in (60), (61) and (62) are, indeed, as claimed, polynomial type or exponential type. ♢

Exercise 6.2: Let T_n be a complete binary tree with $n \geq 1$ nodes. So $n = 2^{h+1} - 1$ where h is the height of T_n. Suppose an algorithm has to visit all the nodes of T_n and at each node of height $i \geq 0$, expend $(i+1)^2$ units of work. Let $T(n)$ denote the total work expended by the algorithm at all the nodes. Give a tight upper and lower bounds on $T(n)$. ♢

Exercise 6.3: (a) Show that the summation $\sum_{i \geq 2}^{n} (\lg n)^{\lg n}$ is neither polynomial-type nor exponential-type.
(b) Estimate this sum. ♢

Exercise 6.4: For this problem, please use arguments from first principles. Do not use calculus, properties of $\log x$ such as $x/\log x \to \infty$, etc. Show that $H_n = o(n^\alpha)$ for any $\alpha > 0$. HINT: Generalize the argument in the text. ♢

Exercise 6.5: Use the method of grouping to show that $S(n) = \sum_{i=1}^{n} \frac{\lg i}{i}$ is $\Omega(\lg^2 n)$. ♢

Exercise 6.6: Give the Θ-order of the following sums: if you use our summation rules, then you must show that the terms has the appropriate growth types.
(a) $S = \sum_{i=1}^{n} \sqrt{i}$.
(b) $S = \sum_{i=1}^{n} \lg(n/i)$. ♢

Exercise 6.7: Let $f(i) = f_n(i) = \frac{i-1}{n-i+1}$. The sum $F(n) = \sum_{i=1}^{n} f_n(i)$ is neither polynomial-type nor exponential-type. Give a Θ-order bound on $F(n)$. HINT: transform this into something familiar. ♢

Exercise 6.8: Can our summation rules for $S(n) = \sum_{i=1}^{n} f(i)$ be extended to the case where $f(i)$ is "decreasing polynomially", suitably defined? NOTE: such a definition must somehow distinguish between $f(i) = 1/i$ and $f(i) = 1/(i^2)$, since in one case $S(n)$ diverges and in the other it converges as $n \to \infty$. ♢

END EXERCISES

§7. Domain Transformation

So our goal for a general recurrence is to transform it into the standard form. You may think of change of domain as a "change of scale". Transforming the domain of a recurrence equation may sometimes bring it into standard form. Consider

$$T(N) = T(N/2) + N. \tag{71}$$

We define

$$t(n) := T(2^n), \quad N = 2^n.$$

This transforms the original N-domain into the n-domain. The new recurrence is now in standard form,

$$t(n) = t(n-1) + 2^n.$$

By DIC, we may choose the boundary condition $t(0) = 0$, we get $t(n) = \sum_{i=0}^{n} 2^i$. This is a geometric series which we know how to sum, $t(n) = 2^{n+1} - 1$; hence, $T(N) = 2N - 1$.

Hey, you should choose $t(0) = 1$ to obtain the more elegant solution $T(N) = 2N$

¶25. Logarithmic transform. More generally, consider the recurrence

$$T(N) = T\left(\frac{N}{c} - d\right) + F(N), \quad c > 1, \tag{72}$$

and d is an arbitrary constant. It is instructive to begin with the case $d = 0$. Consider the "logarithmic transformation" of the argument N to the new argument $n := \log_c(N)$. Then N/c transforms to $\log_c(N/c) = n - 1$. Then $T(N) = T(N/c) + F(N)$ transforms into the new recurrence

So $N = c^n$

$$t(n) = t(n-1) + f(n)$$

where we define

$$t(n) := T(c^n) = T(N), \qquad f(n) := F(N).$$

The preceding manipulation exploits some implicit conventions: $N \leftrightarrow n$, $T \leftrightarrow t$, $F \leftrightarrow f$. This might be confusing in more complicated situations, so let us make the connection between t and T more explicit. Let τ denote the **domain transformation function**,

$$\tau(N) := \log_c(N), \qquad \tau^{-1}(n) = c^n$$

So "n" is a short-hand for "$\tau(N)$" in our convention.

Then $t(\tau(N))$ is defined to be $T(N)$, valid for large enough N. In order for this to be well-defined, we need τ to have an inverse for large enough n. Then we can write

$$t(n) := T(\tau^{-1}(n)).$$

We now return to the general case where d is an arbitrary constant. Note that if $d < 0$ then we must assume that N is sufficiently large (how large?) so that the recurrence (72) is meaningful (*i.e.*, $(N/c) - d < N$). The following "generalized logarithmic transformation"

$$n := \tau(N) = \log_c(N + \frac{cd}{c-1}) \tag{73}$$

will reduce the recurrence to standard form. To see this, note that the inverse transformation is

$$\begin{aligned} N &:= c^n - \frac{cd}{c-1} \\ &= \tau^{-1}(n) \\ (N/c) - d &= c^{n-1} - \frac{d}{c-1} - d \\ &= c^{n-1} - \frac{cd}{c-1} \\ &= \tau^{-1}(n-1). \end{aligned}$$

Writing $t(n)$ for $T(\tau^{-1}(n))$ and $f(n)$ for $F(\tau^{-1}(n))$, we convert equation (72) to

$$\begin{array}{lll} t(n) = T(\tau^{-1}(n)) & & \text{(by definition of } t(n)) \\ = T(N) & & (N = \tau^{-1}(n)) \\ = T((N/c) - d) + F(N) & & \text{(expansion)} \\ = T(\tau^{-1}(n-1)) + F(\tau^{-1}(n)) & & \text{(domain transform)} \\ = t(n-1) + f(n) & & \text{(definition of } t(n) \text{ and } f(n)) \\ = \sum_{i \geq 1}^{n} f(i) & & \text{(telescopy and by DIC)} \end{array}$$

To finally "solve" for $t(n)$ we need to know more about the function $F(N)$. For example, if $F(N)$ is a polynomially bounded function, then $f(n) = F(c^n - \frac{cd}{c-1})$ would be $\Theta(F(c^n))$. This is the justification for ignoring the additive term "d" in the equation (72).

¶26. Division transform. Notice that the logarithmic transform case does not quite capture the following closely related recurrence

$$T(N) = T(N-d) + F(N), d > 0. \tag{74}$$

It is easy to concoct the necessary domain transformation: replace N by $n = N/d$ and substituting

$$t(n) = T(dn)$$

will transform it to the standard form,

$$t(n) = t(n-1) + F(dn).$$

Again, we can explicitly introduce the "division transform" function $\tau(N) = N/d$, etc.

¶27. General Pattern. In general, we consider $T(N) = T(r(N)) + F(N)$ where $r(N) < N$ is some function. We want a domain transform $n = \tau(N)$ so that

$$\tau(r(N)) = \tau(N) - 1. \tag{75}$$

The generalized logarithm transform (73) is of this type. Here is another example: if $r(N) = \sqrt{N}$ we may choose

$$\tau(N) = \lg \lg(N). \tag{76}$$

Then we see that

$$\tau(\sqrt{N}) = \lg(\lg(\sqrt{N})) = \lg(\lg(N)/2) = \lg\lg N - 1 = \tau(N) - 1.$$

Applying this transformation to the recurrence

$$T(N) = T(\sqrt{N}) + N, \tag{77}$$

we may define $t(n) := T(\tau^{-1}(n)) = T(2^{2^n}) = T(N)$, thereby transforming the recurrence (77) to to $t(n) = t(n-1) + 2^{2^n}$.

Note that the transformation (76) may be regarded as two applications of the logarithmic transform. Domain transformation can be confusing because of the difficulty of keeping straight the similar-looking symbols, 'n' versus 'N' and 't' versus 'T'. Of course, these symbols are mnemonically chosen. When properly used, these conventions reduce clutter in our formulas. But if they are confusing, you can always fall back to the use of the explicit transformation functions such as τ.

EXERCISES

Exercise 7.1: Solve recurrence (72) in these cases:
(a) $F(N) = N^k$.
(b) $F(N) = \log N$. ♢

Exercise 7.2: (a) Solve the following four recurrences using domain transformation:

$$T(N) = T(N/2) + \begin{cases} \lg N \\ 1 \\ 1/\lg N \\ 1/\lg^2 N \end{cases}.$$

(b) Generalize the above result: solve the recurrence $T(N) = T(N/2) + \lg^c N$ for all real values of c. ♢

Exercise 7.3: Justify the simplification step (iv) in §1 (where we replace $\lceil n/2 \rceil$ by $n/2$). ♢

Exercise 7.4: Construct examples where you need to compose two or more of the above domain transformations. ♢

END EXERCISES

§8. Range Transformation

A transformation of the range is sometimes called for. For instance, consider

$$T(n) = 2T(n-1) + n.$$

To put this into standard form, we could define

$$t(n) := \frac{T(n)}{2^n}$$

and get the standard form recurrence

$$t(n) = t(n-1) + \frac{n}{2^n}.$$

Telescoping gives us a series of the type in equation (49), which we know how to sum. Specifically, $t(n) = \sum_{x \geq 1}^{n} \frac{x}{2^x} = \Theta(1)$ as $f(x) = x/2^x$ is exponentially decreasing. Hence $T(n) = \Theta(2^n)$.

We have transformed the range of $T(n)$ by introducing a multiplicative factor 2^n: this factor is called the **summation factor**. The reader familiar with linear differential equations will see an analogy with "integrating factor". (In the same spirit, the previous trick of domain transformation is simply a "change of variable".)

In general, a range transformation converts a recurrence of the form

$$T(n) = c_n T(n-1) + F(n) \tag{78}$$

into standard form. Here c_n is a constant depending on n. Let us discover which summation factor will work. If $C(n)$ is the summation factor, we get

$$t(n) := \frac{T(n)}{C(n)},$$

and hence

$$\begin{aligned} t(n) &= \frac{T(n)}{C(n)} \\ &= \frac{c_n}{C(n)} T(n-1) + \frac{F(n)}{C(n)} \\ &= \frac{T(n-1)}{C(n-1)} + \frac{F(n)}{C(n)}, \qquad \text{(provided } C(n) = c_n C(n-1)\text{)} \\ &= t(n-1) + \frac{F(n)}{C(n)}. \end{aligned}$$

Thus we need $C(n) = c_n C(n-1)$ which expands into

$$C(n) = c_n c_{n-1} \cdots c_1.$$

EXERCISES

Exercise 8.1: Z.H. proposed to transform the recurrence $T(n) = 100T(n-1) + f(n)$ by using range transformation $t(n) = T(n)/100$. Convince Z.H. that this is futile. ♢

Exercise 8.2: Solve the recurrence (78) in the case where $c_n = 1/n$ and $F(n) = 1$. ♢

Exercise 8.3: (a) Reduce the following recurrence

$$T(n) = 4T(n/2) + \frac{n^2}{\lg n}$$

to standard form. Then solve it exactly when n is a power of 2.
(b) Extend the solution of part(a) to general n using our generalized Harmonic numbers H_x for real $x \geq 2$ (see §2). You may choose any suitable initial conditions, but please state it explicitly. ♢

Exercise 8.4: Repeat the previous question for the following recurrences:
(a) $T(n) = 4T(n/2) + \frac{n^2}{\lg^2 n}$
(b) $T(n) = 4T(n/2) + \frac{n^2}{\sqrt{\lg n}}$. ♢

END EXERCISES

§9. Differencing and QuickSort

Summation is the discrete analogue of integration. Extending this analogy, we introduce the **differencing** as the discrete analogue of differentiation. As expected, differencing is the inverse of summation. The differencing operation ∇ applied to any complexity function $T(n)$ yields another function ∇T defined by

$$(\nabla T)(n) = T(n) - T(n-1).$$

Differentiation often simplifies an equation: thus, $f(x) = x^2$ is simplified to the linear equation $(Df)(x) = 2x$, using the differential operator D. Similarly, differencing a recurrence equation for $T(n)$ may lead to a simpler recurrence for $(\nabla T)(n)$.

Indeed, the "standard form" (57) can be rewritten as

$$\nabla t(n) = f(n).$$

This is just an equation involving a difference operator — the discrete analogue of a differential equation.

For example, consider the recurrence

$$T(n) = n + \sum_{i=1}^{n-1} T(i).$$

This recurrence does not immediately yield to the previous techniques. But note that

$$(\nabla T)(n) = 1 + T(n-1).$$

Hence $T(n) - T(n-1) = 1 + T(n-1)$ and $T(n) = 2T(n-1) + 1$, which can be solved by the method of range transformation. (Solve it!)

¶28. QuickSort. A well-known application of differencing is the analysis of the QuickSort algorithm of Hoare. In QuickSort, we randomly pick a "pivot" element p. If p is the ith largest element, this subdivides the n input elements into $i-1$ elements less than p and $n-i$ elements greater than p. Then we recursively sort the subsets of size $i-1$ and $n-i$. For a detailed description of QuickSort, including a different analysis, see Lecture VIII. The recurrence is

$$T(n) \quad = \quad n + \frac{1}{n}\sum_{i=0}^{n-1}(T(i-1) + T(n-i)), \tag{79}$$

since for each i, the probability that the two recursive subproblems in QuickSort are of sizes i and $n-i$ is $1/n$. The additive factor of "n" indicates the cost (up to a constant factor) to subdivide the subproblems, and there is no cost in "merging" the solutions to the subproblems. The recurrence (79) is an example of a **full-history recurrence**, so-called because $T(n)$ depends on $T(m)$ for all smaller values of m.

Simplifying (79),

$$\begin{array}{lcll}
T(n) & = & n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & \\
nT(n) & = & n^2 + 2\sum_{i=0}^{n-1} T(i) & \text{[Multiply by } n\text{]} \\
(n-1)T(n-1) & = & (n-1)^2 + 2\sum_{i=0}^{n-2} T(i) & \text{[Substitute } n \text{ by } n-1\text{]} \\
nT(n) - (n-1)T(n-1) & = & 2n - 1 + 2T(n-1) & \text{[Differencing operator for } nT(n)\text{]} \\
nT(n) & = & 2n - 1 + (n+1)T(n-1) & \text{[Simplify]} \\
\frac{T(n)}{n+1} & = & \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{T(n-1)}{n} & \text{[Divide by } n(n+1) \text{ (range transform)]} \\
t(n) & = & \frac{2}{n+1} - \frac{1}{n(n+1)} + t(n-1) & \text{[Define } t(n) = T(n)/(n+1)\text{]} \\
 & = & 2(H_{n+1} - 1) - \sum_{i=1}^{n} \frac{1}{i(i+1)} + t(0) & \text{[Telescoping a standard form]}
\end{array}$$

Thus we see that $t(n) \leq 2H_{n+1}$ (assuming $t(0) = 0$) and hence we conclude

$$T(n) = 2n \ln n + \mathcal{O}(n).$$

It is also easy to get the exact solution for $t(n)$, by evaluating the sum $\sum_{i=1}^{n} \frac{1}{i(i+1)}$ (in a previous Exercise).

¶29. QuickSelect. The following recurrence is a variant of the QuickSort recurrence, and arises in the average case analysis of the QuickSelect algorithm:

$$T(n) = n + \frac{T(1) + T(2) + \cdots + T(n-1)}{n} \tag{80}$$

In the selection problem we need to "select the kth largest" where k is given (This problem is studied in more detail in Lecture XXX). Recursively, after splitting the input set into subsets of sizes $i-1$ and $n-i$ (as in QuickSort), we only need to continue one one of the two subsets (unless the pivot element is already the kth largest that we seek). This explains why, compared to (), the only change in (80) is to replace the constant factor of 2 to 1. To solve this, let us first multiply the equation by n (a range transform!). Then, on differencing, we obtain

$$\begin{array}{rcl}
nT(n) - (n-1)T(n-1) & = & 2n - 1 + T(n-1) \\
nT(n) - nT(n-1) & = & 2n - 1 \\
T(n) - T(n-1) & = & 2 - \frac{1}{n} \\
T(n) & = & 2n - \ln n + \Theta(1).
\end{array}$$

Again, note that we essentially obtain an exact solution.

¶30. Improved QuickSort. We further improve the constants in QuickSort by first randomly choosing three elements, and picking the median of these three to be our pivot. The resulting recurrence is slightly more involved:

$$T(n) = n + \sum_{i=2}^{n-1} p_i [T(i-1) + T(n-i)] \tag{81}$$

where

$$p_i = \frac{(i-1)(n-i)}{\binom{n}{3}}$$

is the probability that the pivot element gives rise to subproblems of sizes $i-1$ and $n-i$.

See Lecture 8 on Probabilistic Analysis where we discuss QuickSort

EXERCISES

Exercise 9.1: Solve the following recurrences to Θ-order:

$$T(n) = n + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i).$$

HINT: Because of the upper bound $\lfloor n/2 \rfloor$, the function $\nabla T(n)$ has different behavior depending on whether n is even or odd. Simple differencing does not seem to work well here. Instead, we suggest the guess and verify-by-induction approach. ♢

Exercise 9.2: Generalize the previous question. Consider the recurrence

$$T(n) = n + \frac{c}{n}\sum_{i=1+\lfloor \alpha n \rfloor}^{n-1} T(i)$$

where $c > 0$ and $0 \leq \alpha < 1$ are constants.
(a) Solve the recurrence for $c = 2$.
(b) Solve $T(n)$ when $c = 4$ and $\alpha = 0$.
(c) Fix $c = 4$. Determine the range of α such that $T(n) = \Theta(n)$. You need to argue why $T(n)$ is not $\Theta(n)$ for α outside this range.
(d) Determine the solution of this recurrence for general c, α. ♢

Exercise 9.3: (a) Suppose that in the base case of QuickSort, we do nothing whenever the size of the subarray to be sorted has 10 or less keys. Call this "`QuirkSort`".
(i) Describe the nature of the output from `QuirkSort`.
(ii) Describe a linear time method to take the output of `QuirkSort` and make it into a sorted array.
(iii) Explain why your method in (ii) takes linear time. ♢

Exercise 9.4:
(a) Show that every polynomial $p(X)$ of degree d can be written as a sum of binomial coefficients with suitable coefficients c_i:

$$p(X) = c_d\binom{X}{d} + c_{d-1}\binom{X}{d-1} + \cdots + c_1\binom{X}{1} + c_0.$$

(b) Assume the above form for $p(X)$, express $(\nabla p)(X)$ as a sum of binomial coefficients. HINT: what is $\nabla\binom{m}{n}$? ♢

END EXERCISES

§10. The Master Theorem

We first look at a recurrence that does fall under our transformation techniques: the **master recurrence** is

$$T(n) = aT(n/b) + f(n) \tag{82}$$

where $a > 0, b > 1$ are constants and $f(n)$ is the forcing (or driving) function. Our goal is to prove the so-called Master Theorem which provides a "cookbook" formula for solutions of the master recurrence.

We have already seen several instances of this recurrence. Another famous one is Strassen's 1969 algorithm for multiplying two $n \times n$ matrices in subcubic time. Strassen's recurrence is $T(n) = 7T(n/2) + n^2$, which has solution $T(n) = \Theta(n^{\lg 7})$. Evidently, the Master recurrence is the recurrence to solve if we manage to solve a problem of size n by breaking it up into a subproblems each of size n/b, and merging these a sub-solutions in time $f(n)$. The recurrence was systematically studied by Bentley, Haken and Saxe [1]. Solving it requires a combination of domain and range transformation.

First apply a domain transformation by defining a new function $t(k)$ from $T(n)$, where $k = \log_b(n)$:

$$t(k) := T(b^k) \quad (\text{for all } k \in \mathbb{R}).$$

Then (82) transforms into

$$t(k) = a\, t(k-1) + f(b^k).$$

Next, transform the range by using the summation factor $1/a^k$. This defines the function $s(k)$ from $t(k)$:

$$s(k) := t(k)/a^k.$$

Now $s(k)$ satisfies a recurrence in standard form:

$$\begin{aligned} s(k) &= \frac{t(k)}{a^k} \\ &= \frac{t(k-1)}{a^{k-1}} + \frac{f(b^k)}{a^k} \\ &= s(k-1) + \frac{f(b^k)}{a^k} \end{aligned}$$

Telescoping, we get

$$s(k) - s(\{k\}) = \sum_{i \geq 1}^{k} \frac{f(b^i)}{a^i},$$

where $\{k\}$ is the fractional part of k (recall that k is real). Using the DIC, we chose the boundary condition

$$s(x) = f(b^x)/a^x, \qquad \text{for } x < 1$$

in order to end up with the simple formula,

$$s(k) = \sum_{i \geq 1}^{k} \frac{f(b^i)}{a^i}. \tag{83}$$

If we like, we can back substitute to get this solution in terms of the original function $T(n)$:

$$\begin{aligned} T(n) &= t(\log_b n) \\ &= a^{\log_b n} s(\log_b n) \\ &= n^{\log_b a} \sum_{i \geq 1}^{\log_b n} \frac{f(b^i)}{a^i}. \end{aligned}$$

This is the general solution to the master recurrence. But it is an open sum, and we need a closed formula. *Now, we cannot proceed any further without knowing the nature of the function f.*

Let us call the function

$$W(n) = n^{\log_b a} \tag{84}$$

the **watershed function** for our recurrence, and $\log_b a$ the **watershed exponent**. The Master Theorem considers three cases for f. These cases are obtained by comparing f to $W(n)$. The easiest case is where f and W have the same Θ-order (CASE(0)). The other two cases are where f grows "polynomially slower" (CASE$(-)$) or "polynomially faster" (CASE$(+)$) than the watershed function.

So the Master Theorem is a trichotomy, like many analysis we have seen so far!

CASE(0) This is when $f(n)$ satisfies

$$f(n) = \Theta(n^{\log_b a}) = \Theta(a^{\log_b n}). \tag{85}$$

Then $f(b^i) = \Theta(a^i)$ and hence

$$s(k) = \sum_{i\geq 1}^{k} f(b^i)/a^i = \Theta(k). \tag{86}$$

CASE$(-)$ This is when $f(n)$ **grows polynomially slower** than the watershed function:

$$f(n) = \mathcal{O}(n^{-\epsilon+\log_b a}), \tag{87}$$

for some $\epsilon > 0$. Then $f(b^i) = \mathcal{O}(b^{i(\log_b a-\epsilon)}) = \mathcal{O}_1(a^i b^{-i\epsilon})$ (using the subscripting notation for $\mathcal{O}$). So $s(k) = \sum_{i\geq 1}^{k} f(b^i)/a^i = \sum \mathcal{O}_1(b^{-i\epsilon}) = \mathcal{O}_2(1)$, since $b > 1$ implies $b^{-\epsilon} < 1$. Hence

$$s(k) = \Theta(1). \tag{88}$$

CASE$(+)$ This is when $f(n)$ satisfies the **regularity condition**

$$af(n/b) \leq cf(n) \text{ (ev.)} \tag{89}$$

for some $c < 1$. Expanding this,

$$\begin{aligned} f(n) &\geq \frac{a}{c} f\left(\frac{n}{b}\right) \\ &\geq \left(\frac{a}{c}\right)^{\log_b n} f(1) \\ &= \Omega(n^{\epsilon+\log_b a}), \end{aligned}$$

where $\epsilon = -\log_b c > 0$. Thus the regularity condition implies that $f(n)$ **grows polynomially faster** than the watershed function,

$$f(n) = \Omega(n^{\epsilon+\log_b a}). \tag{90}$$

It follows from (89) that $f(b^{k-i}) \leq (c/a)^i f(b^k)$. So

$$\begin{aligned} s(k) &= \sum_{i\geq 1}^{k} f(b^i)/a^i \\ &= \sum_{i=0}^{k-1} f(b^{k-i})/a^{k-i} \\ &\leq \sum_{i=0}^{k-1} (c/a)^i f(b^k)/a^{k-i} \\ &= f(b^k)/a^k \left(\sum_{i=0}^{k-1} c^{k-i}\right) \\ &= \mathcal{O}\left(\frac{f(b^k)}{a^k}\right), \end{aligned}$$

since $c < 1$. But clearly, $s(k) \geq f(b^k)/a^k$. Hence we have

$$s(k) = \Theta(f(b^k)/a^k). \tag{91}$$

Summarizing,

$$s(k) = \Theta \begin{cases} 1, & \text{CASE}(-), \quad \text{see (88)}, \\ k, & \text{CASE}(0), \quad \text{see (86)}, \\ f(b^k)/a^k, & \text{CASE}(+), \quad \text{see (91)}. \end{cases}$$

Back substituting using $s(k) = t(k)/a^k$, we get

$$t(k) = a^k s(k) = \Theta \begin{cases} a^k, & \text{CASE}(-) \\ a^k k, & \text{CASE}(0) \\ f(b^k), & \text{CASE}(+). \end{cases}$$

Further back substitution using $T(n) = t(\log_b n)$, we conclude:

THEOREM 7 (Master Theorem). *The master recurrence (82) has solution:*

$$T(n) = \Theta \begin{cases} n^{\log_b a}, & \text{if } f(n) = \mathcal{O}(n^{-\epsilon+\log_b a}), \text{ for some } \epsilon > 0, \quad CASE(-) \\ n^{\log_b a} \log n, & \text{if } f(n) = \Theta(n^{\log_b a}), \quad CASE(0) \\ f(n), & \text{if } af(n/b) \leq cf(n) \text{ for some } c < 1. \quad CASE(+) \end{cases}$$

Informally, we describe CASE$(+)$ as the case when the driving function $f(n)$ is polynomially faster than $W(n)$. But the actual requirement is somewhat stronger, namely the regularity condition (89). In applications of the Master Theorem, this case is usually the least convenient to check.

We can take advantage of the fact that checking if a function $f(n)$ is polynomially faster (or slower) than $W(n)$ is usually easier to check (just by "inspection"). Hence we normally begin by first verifying the polynomially faster condition, equation (90). If so, we then check the stronger regularity condition (89). To illustrate this process, consider the recurrence

$$T(n) = 3T(n/10) + \sqrt{n}/\lg n.$$

We note that $\alpha = \log_{10} 3 < \log_9 3 = 1/2$ and so $n^\alpha \leq \sqrt{n}/\lg n$ (ev.), confirming equation (90). We now suspect that CASE$(+)$ holds, and must verify that

$$cf(n) \geq 3f(n/10)$$

The Master Theorem is powerful but unfortunately, there are gaps between its 3 cases. For instance, $f(n) = n^{\log_b a} \log n$ grows faster than the watershed function, but not polynomially faster. Thus the Master Theorem is inapplicable for this $f(n)$. Yet it is just as easy to solve this case using the transformation techniques (see Exercise). The polynomial version of the theorem is perhaps most useful:

COROLLARY 8. *Let $a > 0, b > 1$ and k be constants. The solution to $T(n) = aT(n/b) + n^k$ is given by*

$$T(n) = \Theta \begin{cases} n^{\log_b a}, & \text{if } \log_b a > k \\ n^k, & \text{if } \log_b a < k \\ n^k \lg n, & \text{if } \log_b a = k \end{cases}$$

What if the values a, b in the master recurrence are not constants but depends on n? For instance, attempting to apply this theorem to the recurrence

$$T(n) = 2^n T(n/2) + n^n$$

(with $a = 2^n$ and $b = 2$), we obtain the false conclusion that $T(n) = \Theta(n^n \log n)$. See Exercises. The paper [18] treats the case $T(n) = a(n)T(b(n)) + f(n)$. For other generalizations of the master recurrence, see [17].

¶31. Graphic Interpretation of the Master Recurrence. We imagine a recursion tree with branching factor of a at each node, and every leaf of the tree is at level $\log_b a$. We further associate a "size" of n/b^i and "cost" of $f(n/b^i)$ to each node at level i (root is at level $i = 0$). Then $T(n)$ is just the sum of the costs at all the nodes. The Master Theorem says this: In case (0), the total cost associated with nodes at any level is $\Theta(n^{\log_b a})$ and there are $\log_b n$ levels giving an overall cost of $\Theta(n^{\log_b a} \log n)$. In case $(+1)$, the cost associated with the root is $\Theta(T(n))$. In case (-1), the total cost associated with the leaves is $\Theta(T(n))$. Of course, this "recursion tree" is not realizable unless a and $\log_b a$ are integers: but it is a useful heuristic for remembering how the Master Theorem works.

Draw the recursion tree with a grain of salt!

¶32. Beyond the Master Theorem. Several authors have extended the Master Theorem to driving function $f(n)$ that has the form $f(n) = W(n) \log^c n$ for all $c \in \mathbb{R}$. This leads to four possible cases:

$$T(n) = \Theta \begin{cases} f(n) & \text{if } f(n) \text{satisfies the regularity condition} & \text{CASE}(+) \\ W(n) \log^{c+1} n & \text{if } c > -1 & \text{CASE}(0) \\ W(n) \log\log n & \text{if } c = -1 & \text{CASE}(1) \\ W(n) & \text{else.} & \text{CASE}(-) \end{cases}$$

We can further generalize this to infinitely many cases, by allow $f(n)$ to be any product of powers of iterated logarithms (these are so-called EL-functions). In the next section, we consider generalizations of another nature.

EXERCISES

Exercise 10.1: Which is the faster growing function: $T_1(n)$ or $T_2(n)$ where

$$T_1(n) = 6T_1(n/2) + n^3, \qquad T_2(n) = 8T_2(n/2) + n^2.$$

♢

Exercise 10.2: Use the Master Theorem to solve the following recurrences arising from matrix multiplication. Be sure to justify the case you choose.

(a) It is easy to see how to recursively multiply two $n \times n$ matrices asymptotically $T(n) = 8T(n/2) + n^2$ time:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} = \begin{bmatrix} aa' + bc' & ab' + bd' \\ ca' + dc' & cb' + dd' \end{bmatrix}$$

What is the solution $T(n)$ using Master theorem?

(b) Strassen (1969) showed that you can actually save one sub-matrix multiplication, giving the recurrence $S(n) = 7S(n/2) + n^2$. Use the Master theorem to determine $S(n)$.

(c) Coppersmith and Winograd (1990) has the current fastest algorithm for matrix multiplication, achieving a bound of $\mathcal{O}(n^{2.376})$ time. Suppose you read in New York Times tomorrow morning, that someone has discovered a marvelous way of multiplying 2×2 matrices using only a multiplications, and the recurrence $T(n) = aT(n/2) + n^2$ yields a faster algorithm than Coppersmith-Winograd. What is the largest possible value of a? What do you think is the likelihood of such a result? ♢

Exercise 10.3: State the Θ-order solution to the following recurrences:

"State" literally means no proofs are needed.

$$\begin{aligned} T(n) &= 10T(n/10) + \log^{10} n. \\ T(n) &= 100T(n/10) + n^{10}. \\ T(n) &= 10T(n/100) + (\log n)^{\log\log n}. \\ T(n) &= 16T(n/4) + 4^{\lg n}. \end{aligned}$$

♢

Exercise 10.4: Solve the following using the Master's theorem.
(a) $T(n) = 3T(n/25) + \log^3 n$
(b) $T(n) = 25T(n/3) + (n/\log n)^3$
(c) $T(n) = T(\sqrt{n}) + n$.
HINT: in the third problem, the Master theorem is applicable after a simple transformation. ♢

Exercise 10.5: Sometimes the Master Theorem is not applicable directly. But it can still be used to yield useful information. Use the Master Theorem to give as tight an upper and lower bound you can for the following recurrences:
(a) $T(n) = n^3 \log^3 n + 8T(n/2)$
(b) $T(n) = n^2/\log\log n + 9T(n/3)$
(c) $T(n) = 4T(n/2) + 3T(n/3) + n$. ♢

Exercise 10.6: Suppose $T(n) = n + 3T(n/2) + 2T(n/3)$. Joe claims that $T(n) = O(n)$, Jane claims that $T(n) = O(n^2)$, John claims that $T(n) = O(n^3)$. Who is closest to the truth? You must justify your answer by appeal to the standard Master Theorem only. ♢

Exercise 10.7: We want to improve on Karatsuba's multiplication algorithm. We managed to subdivide a problem of size n into $a \geq 2$ subproblems of size $n/4$. After solving these a subproblems, we could combine their solutions in $O(n)$ time to get the solution to the original problem of size n. To beat Karatsuba, what is the maximum value a can have? ♢

Exercise 10.8: Suppose algorithm A_1 has running time satisfying the recurrence

$$T_1(n) = aT(n/2) + n$$

and algorithm A_2 has running time satisfying the recurrence

$$T_2(n) = 2aT(n/4) + n.$$

Here, $a > 0$ is a parameter which the designer of the algorithm can choose. Compare these two running times for various values of a. ♢

Exercise 10.9: Suppose

$$T_0(n) = 18T_0(n/6) + n^{1.5}$$

and

$$T_1(n) = 32T_1(n/8) + n^{1.5}.$$

Which is the correct relation: $T_0(n) = \Omega(T_1(n))$ or $T_0(n) = \mathcal{O}(T_1(n))$? We want you to do this exercise without using a calculator or its equivalent; instead, use inequalities such as $\log_8(x) < \log_6(x)$ (for $x > 1$) and $\log_6(2) < 1/2$. ♢

Exercise 10.10: How is the regularity condition on $f(n)$ and the condition that $f(n)$ increase polynomially related? What can you say about the sum $\sum_{i=1}^{n} f(i)$ when f satisfies the regularity condition for some a, b, c? ♢

Exercise 10.11: Solve the master recurrence when $f(n) = n^{\log_b a} \log^k n$, for any $k \geq 1$. ♢

Exercise 10.12: Show that the master theorem applies to the following variation of the master recurrence:

$$T(n) = a \cdot T(\frac{n+c}{b}) + f(n)$$

where $a > 0, b > 1$ and c is arbitrary. ♢

Exercise 10.13:

(a) Solve $T(n) = 2^n T(n/2) + n^n$ by direct expansion.

(b) To what extent can you generalize the Master theorem to handle some cases of $T(n) = a_n T(n/b_n) + f(n)$ where a_n, b_n are both functions of n? ♢

Exercise 10.14: Let $W(n)$ be the watershed function of the master recurrence. In what sense is the "watershed function" of the next order equal to $W(n)/\ln n$? ♢

Exercise 10.15:

(a) Let

$$s(n) = \sum_{i=1}^{n} \frac{\lg i}{i}$$

Prove that $s(n) = \Theta(\lg^2 n)$. For the lower bound, we want you to use real induction, and the fact that for $n \geq 2$, we have

$$\ln(n) - (2/n) < \ln(n-1) < (\ln n) - (1/n).$$

(b) Using the domain/range transformations to solve the following recurrence:

$$T(n) = 2T(n/2) + n\frac{\lg\lg n}{\lg n}.$$

♢

Exercise 10.16: Consider the recurrence $T(n) = aT(n/b) + \frac{n^4}{\log n}$ where $a > 0$ and $b > 1$. Describe the set S of all pairs (a, b) for which the Master Theorem gives a solution for this recurrence. Do not describe the solutions. You must describe the set S in the simplest possible terms. ♢

Exercise 10.17: The following recurrences arises in the analysis of a parallel algorithm for hidden-surface removal (Reif and Sen, Proc. ACM Symp. on Comp. Geometry, 1988):

$$T(n) = T(2n/3) + \lg n \lg\lg n$$

Another version of the algorithm [18] leads to

$$T(n) = T(2n/3) + (\lg n)/\lg\lg n.$$

Solve for $T(n)$ in both cases. ♢

END EXERCISES

§11. The Multiterm Master Theorem

The Master recurrence (82) can be generalized to the following **multiterm master recurrence**:

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T\left(\frac{n}{b_i}\right) \tag{92}$$

where $k \geq 1$, $a_i > 0$ (for all $i = 1, \ldots, k$) and $b_1 > b_2 > \cdots > b_k > 1$. When $k = 2$, we have the following examples of 2-term master recurrences:

$$T(n) = T(c_1 n) + T(c_2 n) + n, \qquad (c_1 + c_2 < 1). \tag{93}$$

$$T(n) = T(n/2) + T(n/4) + \log n. \tag{94}$$

The first recurrence (93) arise in linear time selection algorithms (see Chapter XI). There are many versions of this algorithm with different choices for the constants c_1, c_2. E.g., $c_1 = 7/10, c_2 = 1/5$. The second recurrence arose in Computational Geometry. Edelsbrunner and Welzl [3] introduced a data structure called **conjugation tree** for solving the **point retrieval problem**. The exercises will go over this data structure.

¶33. Reducing multiterm to single term master recurrence. Before providing the general solution, let us see how our previous techniques would fare here. First of all, rote expansion seems hopeless, even for a two-term master recurrence. On a more positive note, the method of real induction can provide us with confirmations of guessed upper and lower bounds – we had already seen such examples. The catch is how do we go about guessing these bounds. But here is an interesting method to use the Master Theorem to provide upper and lower bounds. The idea is to convert our multiterm recurrence into a master recurrence: let $a := \sum_{i=1}^{k} a_i$, $b := \min\{b_i : i = 1, \ldots, k\}$, and $c := \max\{b_i : i = 1, \ldots, k\}$. This defines two master recurrences

The student is invited to expand the 2-term recurrences...

$$U(n) = f(n) + aU(n/b), \tag{95}$$

$$L(n) = f(n) + aL(n/c). \tag{96}$$

Clearly, $T(n) = O(U(n))$ and $T(n) = \Omega(L(n))$. Then the Master Theorem implies the bound

$$T(n) = \begin{cases} \mathcal{O}(f(n)\log n + n^{\log_b a}), \\ \Omega(f(n) + n^{\log_c a}). \end{cases} \tag{97}$$

Applying this to the conjugation tree recurrence (94), we obtain

$$T(n) = \begin{cases} \mathcal{O}(n), \\ \Omega(\sqrt{n}). \end{cases}$$

But suppose we first expand our recurrence once:

$$\begin{aligned} T(n) &= \boxed{T(n/2)} + T(n/4) + \log n \\ &= \boxed{T(n/4) + T(n/8) + \log(n/2)} + T(n/4) + \log n \\ &= 2T(n/4) + T(n/8) + \Theta(\log n). \end{aligned}$$

Now the application of (97) yields the sharper bound:

$$T(n) = \begin{cases} \mathcal{O}(n^{\log_4 3}), \\ \Omega(n^{\log_8 3}). \end{cases}$$

It is clear that this trick can be repeated. We remark that the lower bound can sometimes be improved by omitting terms before taking the maximum to form c. E.g., for $T(n) = T(n/2) + T(n/3) + T(n/9) + 1$, the above scheme yields $T(n) = \Omega(\sqrt{n})$, but if we first drop the term $T(n/9)$, we get the improvement $T(n) = \Omega(n^{\log_3 2})$.

¶34. Multiterm Generalization of Master Theorem. To state the multiterm analogue of the Master Theorem, we must generalize two concepts from the Master Theorem: (a) Associated with the recurrence (92) is the **watershed constant**, a real number α such that

$$\sum_{i=1}^{k} \frac{a_i}{b_i^\alpha} = 1. \tag{98}$$

Clearly α exists and is unique since the sum (98) tends to 0 as $\alpha \to \infty$, and tends to ∞ as $\alpha \to -\infty$. As usual, let $W(n) = n^\alpha$ denote the watershed function. (b) The recurrence (92) gives rise to a **generalized regularity condition** on the driving (or forcing) function $f(n)$, namely,

$$\sum_{i=1}^{k} a_i f(n/b_i) \leq c f(n) \tag{99}$$

for some $0 < c < 1$.

THEOREM 9 (Multiterm Master Theorem).

$$T(n) = \Theta \begin{cases} n^\alpha \log n & \text{if } f(n) = \Theta(n^\alpha) \\ n^\alpha & \text{if } f(n) = \mathcal{O}(n^{\alpha-\varepsilon}), \text{for some } \varepsilon > 0, \\ f(n) & \text{if } f \text{ satisfies the regularity condition (99).} \end{cases}$$

Before proving this result, let us see its application to the conjugation tree recurrence (94). The watershed constant α satisfies the equation $\frac{1}{2^\alpha} + \frac{1}{4^\alpha} = 1$. Writing $x = \frac{1}{2^\alpha}$, we get the equation $x + x^2 = 1$. The positive solution to this quadratic equation is $x = 2^{-\alpha} = (-1+\sqrt{5})/2$. This yields $\alpha = \lg(-1+\sqrt{5}) - 1 \sim 0.695$. Edelsbrunner and Welzl obtained α by using an analogy with Fibonacci recurrences, but we now see that it can be systematically derived. They proved that $T(n) = O(n^\alpha)$; our theorem further shows that their bound is Θ-tight.

Proof of Multiterm Master Theorem. We use real induction.

CASE(0): Assume that $f(n) = \Theta_1(W(n))$. We will show that $T(n) = \Theta_2(W(n)\log n)$. We have

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &= \Theta_1(n^\alpha) + \textstyle\sum_{i=1}^k a_i \Theta_2\left(\left(\frac{n}{b_i}\right)^\alpha \log\left(\frac{n}{b_i}\right)\right) && \text{(by induction)} \\ &= \Theta_1(n^\alpha) + \Theta_2(n^\alpha)\left[\textstyle\sum_{i=1}^k \frac{a_i}{b_i^\alpha} \log\left(\frac{n}{b_i}\right)\right] \\ &= \Theta_1(n^\alpha) + \Theta_2(n^\alpha)\left[\log n - D\right], && \text{(where } D = \textstyle\sum_{i=1}^k \frac{a_i}{b_i^\alpha}\log(b_i) \text{ and using (98))} \\ &= \Theta_2(n^\alpha \log n). \end{aligned}$$

Let us elaborate on the last equality. Suppose $f(n) = \Theta_1(n^\alpha)$ amounts to the inequalities $c_1 W(n) \leq f(n) \leq C_1 W(n)$ (ev.). We must choose c_2, C_2 such that $c_2 W(n)\log n \leq T(n) \leq C_2 W(n)\log n$ (ev.). The following choice suffices:

$$C_2 = C_1/D, \qquad c_2 = c_1/D.$$

CASE($-$): Assume $0 \leq f(n) \leq D_1 n^{\alpha-\varepsilon}$ for some $\varepsilon > 0$. The lower bound is easy: assume $T(n/b_i) \geq c_1(n/b_i)^\alpha$ (ev.) for each i. Then[6]

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &\geq \textstyle\sum_{i=1}^k a_i c_1 (\frac{n}{b_i})^\alpha && \text{(since } f(n) \geq 0 \text{ and by induction)} \\ &= c_1 n^\alpha. \end{aligned}$$

[6] The fact $f(n) \geq 0$ (ev.) is a consequence of "$f \in \mathcal{O}(n^{\alpha-\varepsilon})$" and the definition of the big-Oh notation.

The upper bound needs a slightly stronger hypothesis: assume $T(n/b_i) \leq C_1 n^\alpha (1 - n^{-\varepsilon})$ (ev.). Then

$$\begin{array}{rcll} T(n) & = & f(n) + \sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) & \\ & \leq & D_1 n^{\alpha-\varepsilon} + \sum_{i=1}^k a_i C_1 \left(\frac{n}{b_i}\right)^\alpha \left[1 - \left(\frac{n}{b_i}\right)^{-\varepsilon}\right] & \text{(by induction)} \\ & = & C_1 n^\alpha - C_1 n^{\alpha-\varepsilon}\left[\sum_{i=1}^k \frac{a_i}{b_i^{\alpha-\varepsilon}} - D_1/C_1\right] & \\ & \leq & C_1 n^\alpha - C_1 n^{\alpha-\varepsilon} & \end{array}$$

provided $\sum_{i=1}^k a_i/b_i^{\alpha-\varepsilon} \geq 1 + (D_1/C_1)$. Since $\sum_{i=1}^k a_i/b_i^{\alpha-\varepsilon} > 1$, we can certainly choose a large enough C_1 to satisfy this.

CASE(+): The lower bound $T(n) = \Omega(f(n))$ is trivial. As for upper bound, assuming $T(m) \leq D_1 f(m)$ (ev.) whenever $m = n/b_i$,

$$\begin{array}{rcll} T(n) & = & f(n) + \sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) & \\ & \leq & f(n) + \sum_{i=1}^k a_i D_1 f(n/b_i) & \text{(by induction)} \\ & = & f(n) + D_1 c f(n) & \text{(by regularity)} \\ & \leq & D_1 f(n) & \text{(if } D_1 \geq 1/(1-c)\text{)} \end{array}$$

This concludes the proof of the Multiterm Master Theorem.

The use of real induction appears to be necessary in this proof: unlike the master recurrence, the multiterm version does not yield to transformations. Again, the generalized regularity condition implies that $f(n) = \Omega(n^{\alpha+\varepsilon})$ for some $\varepsilon > 0$. This is shown by induction:

$$\begin{array}{rcll} f(n) & \geq & \frac{1}{c}\sum_{i=1}^k a_i f(n/b_i) & \\ & \geq & \frac{1}{c}\sum_{i=1}^k a_i D (n/b_i)^{\alpha+\varepsilon} & \text{(by induction, for some } D > 0\text{)} \\ & = & \frac{D}{c} n^{\alpha+\varepsilon} \sum_{i=1}^k \frac{a_i}{b_i^{\alpha+\varepsilon}} & \\ & = & D n^{\alpha+\varepsilon} & \text{(if we choose } c = \sum_{i=1}^k \frac{a_i}{b_i^{\alpha+\varepsilon}}\text{)} \end{array}$$

Since $\sum_{i=1}^k \frac{a_i}{b_i^\alpha} = 1$, we should be able to choose a $\varepsilon > 0$ to satisfy the last condition. Note that this derivation imposes no condition on D, and so D can be determined based on the initial conditions.

EXERCISES

Exercise 11.1: Recall the 2-term recurrence from the analysis of conjugation tree:

$$T(n) = T(n/2) + T(n/4) + \lg n.$$

Numerically determine the watershed constant α in this recurrence. Show α up to 5 decimal places. We don't presume any particular way to do this, except that you can only use an ordinary scientific calculator. Tell us how you obtained your constant. ♢

Exercise 11.2: To understand the recurrence $T(n) = T(n/2) + T(n/3) + T(n/4) + n$, we will explore numerically the function $h(x) = 2^{-x} + 3^{-x} + 4^{-x}$. We want to determine the α such that $h(\alpha) = 1$. For a simple way to do this, use a user-friendly, powerful software like MATLAB. For instance, consider the following two lines of MATLAB code:

```
>>  h = @(x) 2.^(-x) + 3.^(-x) + 4.^(-x);
>>  for x = 0.9 : 0.1 : 1.2, display([x, h(x)]), end
```

The first line defines the function $h(x)$. The second line is a for-loop where x begins with the value 0.9 and each iteration increases the value of x by 0.1 until $x = 1.2$. Each iteration simply prints the pair $(x, h(x))$ of values. This loop produces the values shown in the first of the following four tables:

x	$h(x)$
0.9000	1.1951
1.0000	1.0833
1.1000	0.9828
1.2000	0.8923

x	$h(x)$
1.0700	1.0119
1.0800	1.0021
1.0900	0.9924
1.1000	0.9828

x	$h(x)$
1.0810	1.0011
1.0820	1.0001
1.0830	0.9992
1.0840	0.9982

x	$h(x)$
1.0820	1.0001
1.0821	1.0000
1.0822	0.9999
1.0823	0.9998

By changing the stepsize and limits of the for-loop, we can get more correct digits with run of the for-loop. Each successive table above is obtained this way, each time giving us an extra digit in the decimal expansion of α. Thus, $\alpha \approx 1.0821$. How would you continue this experiment to determine the first 100 digits of α? ♢

Exercise 11.3: Let $T(n) = 2T(n/3) + T(n/10) + 1$. Use the Master Theorem to derive a sublinear upper bound on $T(n)$. ♢

Exercise 11.4: In the text, we sharpened our bounds for the conjugation tree recurrence function $T(n)$ by expanding the recurrence (94) just once, and then applying (97),
(a) Let us now expand (94) twice before applying (97). Verify that the new bounds are further improvements.
(b) Show that this improvement be repeated indefinitely? ♢

Exercise 11.5: Consider $T(n) = T(n/b_1)+T(n/b_2)+T(n/b_3)+1$ where $1 < b_1 \leq b_2 \leq b_3$. What is the lower bound on $T(n)$ using (97)? Under what conditions on b_1, b_2, b_3 can you obtain a better bound by omitting the smallest term? ♢

END EXERCISES

§12. Other Recurrences

There is a wide variety of recurrences which we have barely hinted at. For instance, the typical recurrences arising in counting combinatorial structures have an exponential (e.g., $T(n) = 2T(n-1)+f(n)$) or double exponential growth (e.g., $T(n) = T(n-1)^2 + f(n)$). We refer to Knuth for such examples. In this section, we focus on some other types of recurrences.

§12.1. Recurrences with Max or Min

Many recurrences in computer science involve the Max or Min operation. Here we give three examples.

¶35. QuickSort Variant. Consider the following variant of QuickSort: each time after we partition the problem into two subproblems, we will solve the subproblem that has the smaller size first (if their sizes are equal, it does not matter which order is used). We want to analyze the depth of the recursion stack. If a problem of size n is split into two subproblems of sizes n_1, n_2 then $n_1 + n_2 = n - 1$. Without loss of generality, let $n_1 \leq n_2$. So $0 \leq n_1 \leq \lfloor (n-1)/2 \rfloor$. If the stack contains problems of sizes $(n_1 \geq n_2 \geq \cdots \geq n_k \geq 1)$ where n_k is the problem size at the top of the stack, then we have

$$n_{i-1} \geq n_i + n_{i+1}.$$

Since $n_1 \leq n$, this easily implies $n_{2i+1} \leq n/2^i$ or $k \leq 2 \lg n$. A tighter bound is $k \leq \log_\phi n$ where $\phi = 1.618\ldots$ is the golden ratio. This is not tight either.

The depth of recursion satisfies

$$D(n) = \max_{n_1=0}^{\lfloor (n-1)/2 \rfloor} [\max\{1 + D(n_1), D(n_2)\}]$$

This recurrence involving max is actually easy to solve. Assuming $D(n) \leq D(m)$ for all $n \leq m$, and for any real x, $D(x) = D(\lfloor x \rfloor)$, it is easy to see that $D(n) = 1 + D(n/2)$. Using the fact that $D(1) = 0$, we obtain $D(n) \leq \lg n$. [Note: $D(1) = 0$ means that all problems on the stack has size ≥ 2.

¶36. Solving a Problems on a Binary Tree. Consider this recurrence which involves both Max and Min:

$$C(n) = \max_{m=0,\ldots,n-1} \{C(m) + C(n-m-1) + \min\{m+1, n-m\}\} \tag{100}$$

This represent the cost to solve a recursive problem represented by a binary tree T on n nodes, where the left and right subtrees have sizes m and $n-m-1$, respectively. To solve the problem on T, we recursively solving the problem on the left and right subtrees, and then marry the two sub-solutions at a cost of $\min\{m+1, n-m\}$. We claim that

$$C(n) \leq KnH_n \tag{101}$$

where H_n is the nth Harmonic number and $K \geq 1$ is sufficiently large. By DIC, we can assume (101) is true for all $n \leq n_0$ (for some $n_0 \geq 1$). Inductively, for $n > n_0$, we have

$$C(n) \leq KmH_m + K(n-m-1)H_{n-m-1} + \min\{m+1, n-m\} \tag{102}$$

for some $m = 0, \ldots, n-1$. Note that $m \leftrightarrow n-m-1$ are interchangeable in the RHS of (102). Hence wlog, assume $m \geq n-m-1$. Then $T(n) < KnH_m + n - m$. But $n(H_n - H_m) = \sum_{i=1}^{n-m} \frac{n}{i+m} \geq n-m$. Therefore $T(n) \leq KnH_m + n(H_n - H_m) \leq KnH_n$ (since $K \geq 1$).

This proves $C(n) = O(n \log n)$. This bound exploits the Min in (100). For instance, if we replace the Min by a Max, then the solution is $C(n) = \Theta(n^2)$ (Exercise). We find this $O(n \log n)$ solution instructive: in effect, it says that the worst case value of m in (100) is when $m \sim n/2$, thus reducing the recurrence to look like $C(n) = 2C(n/2) + n$, yielding the $\Theta(n \log n)$ solution. So the Min has the effect of ensuring that the balanced binary tree T is the worst case solution.

Fredman [6] considered the general class of recurrences of the form

$$M(n) = g(n) + \min_{0 \leq k \leq n-1} \{\alpha M(k) + \beta M(n-k-1)\}$$

which arises from analysis of binary search trees.

¶37. Analysis of ϵ-Nets. The following recurrence arise in the analysis of a class of data structures called ϵ-nets, first studied by Haussler and Welzl. Assuming $0 < \epsilon < 1$ and $m \geq 2$ are fixed,

$$T(n) = 1 + \max_{(n_1,\ldots,n_m)} \sum_{i=1}^{m} T(n_i) \tag{103}$$

where the maximum ranges over all $(n_1, \ldots, n_m)$ satisfying $n_i \geq 0$ and $\sum_{i=1}^{m} n_i \leq \epsilon n$. There is a trivial solution to this: the constant function

$$T(n) = 1/(1-m)$$

for all n. But $T(n) < 0$ in this case and we seek a non-negative solution. Assuming that $T(n)$ is a convex cap[7], it is easy to see that

$$T(n) = 1 + mT(\epsilon n/m) = \Theta(n^{\log_{m/\epsilon} m}).$$

To show $T(n)$ is a convex cap, we note that it is continuous (Exercise) and a monotonic non-decreasing function. Then it suffices (Exercise) to prove that

$$T(x) + T(y) \leq 2T((x+y)/2) \tag{104}$$

where we now regard $T(x)$ as a real function defined for all $x \geq 0$. This turns out to be easy to show inductively, assuming the base case where $T(x) = x$ (or $T(x) = 0$) for all $0 \leq x \leq 1$.

§12.2. A Log-square Solution

Consider the recurrence

$$T(n) = 1 + T(n - \frac{n}{\log n}). \tag{105}$$

This does not yield to our standard techniques. To probe deeper, note some simple bounds. It is easy to see that $T(n) \leq n$ since this is the solution to the recurrence $T(n) \leq 1 + T(n-1)$. Likewise $T(n) \geq \lg n$ since this is the solution to $T(n) \geq 1 + T(n/2)$.

To get a better upper bound, we note that

$$\begin{aligned} T(n) &= 1 + T\left(n\left(1 - \frac{1}{\log n}\right)\right) \\ &\leq 2 + T\left(n\left(1 - \frac{1}{\log n}\right)^2\right), \qquad (why?) \\ &\vdots \\ &\leq k + T\left(n\left(1 - \frac{1}{\log n}\right)^k\right) \end{aligned}$$

using monotonicity of $T(n)$. Hence $T(n) = k$ if we assume $T(n) = 0$ for $n \leq 1$ and k is chosen so that

$$\left(1 - \frac{1}{\log n}\right)^k \leq 1/n < \left(1 - \frac{1}{\log n}\right)^{k+1}.$$

[7] We say a real function $f(x)$ is **convex cap** if for all $0 < \alpha < 1$, $f(x) + f(y) \leq 2f(\alpha x + (1-\alpha)y)$. For completeness, we say $f(x)$ is **convex cup** if for all $0 < \alpha < 1$, $f(x) + f(y) \geq 2f(\alpha x + (1-\alpha)y)$.

Taking natural logs, and assuming for simplicity that $\log = \ln$ in (105), we see that

$$\begin{aligned}(k+1)\ln\left(1-\frac{1}{\ln n}\right) &> -\ln n,\\ (k+1)\left(-\frac{1}{\ln n}\right) &> -\ln n, \qquad (\text{since } \ln(1+x) \le x \text{ for } |x|<1),\\ k+1 &< \ln^2 n.\end{aligned}$$

Up to a constant factor, this is also the lower bound: we show that $T(n) \ge C\ln^2 n$ by induction:

$$\begin{aligned}T(n) &\ge 1 + C\ln^2\left(n\left(1-\frac{1}{\log n}\right)\right)\\ &= 1 + C(\ln n + \ln\left(1-\frac{1}{\log n}\right))^2\\ &\ge 1 + C(\ln n - \frac{2}{\ln n})^2, \qquad \text{since } \ln(1+x) \ge x - x^2/2 \text{ for } |x|<1\\ &\ge C\ln^2 n.\end{aligned}$$

Thus $T(n) = \Theta(\ln^2 n)$.

REMARK: If we were told from the beginning to verify that $T(n) = \Theta(\ln^2 n)$, this would be routine. What we are demonstrating here is the process of discovering that $\Theta(\ln^2 n)$ is the correct answer.

EXERCISES

Exercise 12.1: Solve for $C(n)$ where

$$C(n) = \max_{m=0,\ldots,n-1}\{C(m) + C(n-m-1) + \max\{m+1, n-m\}\}.$$

Note that this is similar to (100) except that the Min has been replaced by a Max. ♢

Exercise 12.2: Try to obtain tight constants for the recurrence (105). What if $\log$ is not the natural logarithm in the original equation? ♢

Exercise 12.3: Show that $T(x)$ in (104) is continuous by exploiting the fact that the addition and maximum functions are continuous. ♢

Exercise 12.4: Prove that if $T(x)$ is continuous and satisfies equation (104) then it is a convex cap. ♢

Exercise 12.5: Bound the solution to the recurrence $T(n) = T(n-1) + 2T(n/2) + n$. This is an interesting mixture of linear recurrence and the master recurrence. ♢

Exercise 12.6: (Leighton 1996) Show that $T(n) = 2T(\frac{n}{2} - \frac{n}{\lg n})$ has solution $T(n) = \Theta(n\log^{\Theta(1)} n)$. Assume that $T(n) = 1$ for $n \le 5$, and the recurrence holds for $n > 5$. Thus $T(5+\varepsilon) = 2$, so this function is discontinuous. ♢

Exercise 12.7: Analyze the behavior of the function $T(n)$ defined by the recurrence $T(n) = nT(\log n)$. Give upper and lower bounds for $T(n)$ using "closed form expressions" in terms of the functions $\log^{(i)} n$, $i \geq 0$. **Note:** This recurrence arises from an early version of the fast integer multiplication algorithm of Schönhage and Strassen. ♢

Exercise 12.8: Solve the recurrence $T(n) = 1 + \max_{(n_1,n_2,n_3,n_4)}\{T(n_1) + T(n_2) + T(n_3) + T(n_4)\}$ where $(n_1, \ldots, n_4)$ ranges over all non-negative numbers such that $\sum_{i=1}^{4} n_i = \frac{3n}{2}$ and each $n_i \leq n/2$. ♢

Exercise 12.9: Solve the following recurrences to Θ-order:
(a) $T(n) = 1 + 2T(n - \frac{n}{\log n})$.
(b) $T(n) = 2^n T(n/2) + n^n$.
(c) $T(n) = 1 + T(\frac{n}{\log n})$.
HINT: these recurrences are considerably harder than most of what we encounter. First guess non-tight upper and lower bounds and verify by induction. Then try to tighten these bounds. ♢

END EXERCISES

§12.3. Multivariable Recurrences

So far, our recurrences involve only one variable. But multivariable recurrences arise in several ways: one source of such recurrences is multidimensional problems in computational geometry (one of the variable is the dimension).

The pre-processing problem of **point dominance queries** in d-dimensions is as follows: given a set $S \subseteq \mathbb{R}^d$ of n points, construct a data structure $D(S)$ such that for any query point $p \in \mathbb{R}^d$, we can quickly determine if there is any point $x \in S$ that **dominates** p (this means $x \geq p$, componentwise). One solution is to pick some $c \in R$ such that S splits into two subsets S_1, S_2 of size $n/2$ each, where the first component of each $x \in S_1$ is $\leq c$, and the first component of each $x' \in S_2$ is $\geq c$. To answer the query for p, begin by comparing the first component p_1 of p to c: if $p_1 > c$ then it is sufficient to recursively check if some $x \in S_2$ dominates p. If $p_1 \leq c$, we must do two searches: (i) check if some $x \in S_1$ dominates p and (ii) check if some $x \in S_2$ dominates p. The search in (i) is, however, done in $d - 1$ dimensions since we may ignore the first components. Thus the time for answering queries satisfies the recurrence

$$T(n, d) = 1 + T(n/2, d) + T(n/2, d - 1).$$

It is not hard to see that $T(n, 1) = \mathcal{O}(1)$. Then we may verify the solution $T(n, d) = \Theta(\log^{d-1} n)$.

¶38. Output-sensitive algorithms. Multivariable recurrences arise in the analysis of "output-sensitive" algorithms. Such algorithms has, besides the traditional **input parameter** n, an (implicit) **output parameter** h, which is the measures the size of the output for the given input instance. The computational complexity of such algorithms depends on both n and h. An example is the problem of computing the convex hull of a set of n points in the plane. The output size is just the number of points in the actual convex hull. There are well-known $\mathcal{O}(n \log n)$ algorithms for this problem. Kirkpatrick and Seidel has given an algorithm whose time complexity satisfies the following recurrence:

$$T(n, h) = \mathcal{O}(n) + \max_{h_1+h_2=h-1} \left\{ T(\frac{n}{2}, h_1) + T(\frac{n}{2}, h_2) \right\}.$$

Here, h_i are positive integers. We may assume $T(n,h) = \mathcal{O}(n)$ for $h \leq 3$. To see that $T(n,h) = \mathcal{O}(n \log h)$, we could of course just substitute and verify. But it is more instructive to argue as follows: consider a "recursion tree" corresponding to a possible expansion of the recurrence relation for $T(n,h)$. There are exactly h nodes in this binary tree, where each internal node at depth i (the root is depth 0) carries a "cost" of $n/2^i$. The "cost" of the tree just the sum of these costs at the internal nodes. So $T(n,h)$ is the maximum cost over all possible recursion trees. The *claim* $T(n,h) = \mathcal{O}(n \log h)$ follows if we prove that the maximum cost occurs when the tree has depth at most $\log_2 h$ (since the total cost of all nodes at any depth i is invariably n). For the sake of contradiction, suppose we have a maximum cost tree with depth $d > \log_2 h$. Then there is a node at depth $d-1$ whose children are leaves at depth d. We can transfer these two children to become the children of some other node at depth $\leq d-2$. This would increase the cost for the tree, contradiction.

EXERCISES

Exercise 12.10: Show that if $S(n,d)$ is the space requirement for the above data structure, then $S(n,d) = 1 + 2S(n/2,d) + S(n/2,d-1)$. Solve this recurrence. What is $S(n,1)$? ♢

Exercise 12.11: Consider the following recurrence

$$T(n,h) = \mathcal{O}(n) + \max_{h_1+h_2=h-1; c_1+c_2=1} \{T(c_1 n, h_1) + T(c_2 n, h_2)\}.$$

(a) Solve for $T(n,h)$ with only the assumption $h_i \geq 1, c_i > 0$ in the above.
(b) Solve for $T(n,h)$ with the *additional* assumption that $c_i \leq \alpha$ where $0 < \alpha < 1$ is fixed. Generalize the above argument about the shape of the maximum cost recursion tree. ♢

Exercise 12.12: (Sharir-Welzl) The following recurrence arises in analyzing the diameter of n-dimensional polytopes with m facets:

$$f(n,m) = f(n-1,m-1) + \frac{2}{m}\sum_{i=1}^{m} f(n-1,i).$$

Solve the recurrence. ♢

END EXERCISES

§13. Orders of Growth

The reader should first review the basic properties of the exponential and logarithm functions in the appendix.

Learning to judge the growth rates of complexity functions is a fundamental skill in algorithmics. This section is a practical one, designed to help students develop this skill.

Most complexity functions in practice are the so-called **logarithmico-exponential functions** (for short, L-functions): such functions $f(x)$ are real and defined for all $x \geq x_0$ for some x_0 depending

of f. An L-function is either the identity function x or a constant $c \in \mathbb{R}$, or else obtained as a finite composition with the functions

$$A(x), \qquad \ln(x), \qquad e^x$$

where $A(x)$ denotes a real branch of an algebraic function. For instance, $A(x) = \sqrt{x}$ is the function that picks the real square-root of x. The reader may have noticed that all the common complexity functions are totally ordered in the sense that for any f, g, either $f \preceq g$ or $g \preceq f$. A theorem[8] of Hardy [9] confirms this: *if f and g are L-functions then $f \leq g$ (ev.) or $g \leq f$ (ev.).* In particular, each L-function f is eventually non-negative, $0 \leq f$ (ev.), or non-positive, $f \leq 0$ (ev.).

The following are the common categories of functions you will encounter:

CATEGORY	SYMBOL	EXAMPLES
vanishing term	$o(1)$	$\frac{1}{n}$, 2^{-n}
constants	$\Theta(1)$	1, $2 - \frac{1}{n}$
polylogs	$\log^k n$ (for any $k > 0$)	H_n, $\log^2 n$
polynomials	n^k (for any $k > 0$)	n^3, $\sqrt{n}$
non-polynomials	$n^{\Omega(1)}$	$n!$, 2^n, $n^{\log\log n}$

Note that $n!$ and H_n are not L-functions, but they can be closely approximated by L-functions. The last category forms a grab-bag of anything growing faster than a polynomial. These 5 categories form a hierarchy of increasingly larger Θ-order.

¶39. Rules for comparing functions. We are interested in comparing functions up to their Θ-order. The trick of comparing two functions by taking their logarithms is this: *if $\log f \preceq \log g$ then clearly $f \preceq g$.* But students often think the converse is also true.

We list some simple rules. Most comparisons of interest to us can be reduced to repeated applications of these rules:

Sum: In a direct comparison involving a sum $f(n) + g(n)$, ignore the smaller term in this sum.
E.g., given $n^2 + n \log n + 5$, you should ignore the "$n \log n + 5$" term. However, beware that if the sum appears in an exponent, the neglected part may turn out be decisive when the dominant terms are identical.

Product: If $0 \preceq f \preceq f'$ and $0 \preceq g \preceq g'$ then $fg \preceq f'g'$. (If, in addition, $f \prec f'$ then we have $fg \prec f'g'$.)
E.g., this rule implies $n^b \prec n^c$ when $b < c$ (since $1 \prec n^{c-b}$, by the logarithm rule next).

Logarithm: $1 \prec \log^{(k+1)} n \prec (\log^{(k)} n)^c$ for any integer $k \geq 0$ and real $c > 0$. Here $\log^{(k)} n$ refers to the k-fold application of the logarithm function and $\log^{(0)} n = n$.

Exponentiation: If $1 \leq f \leq g$ (ev.) then $d^f \preceq d^g$ for any constant $d > 1$. If $1 \leq f \leq cg$ (ev.) for some $c < 1$ then $d^f \prec d^g$.

¶40. Example. Suppose we want to compare $n^{\log n}$ versus $(\log n)^n$. By the rule of exponentiation, $n^{\log n} \prec (\log n)^n$ follows if we take logs and show that $\log^2 n \leq 0.5n \log\log n$ (ev.). In fact, we show the stronger $\log^2 n \prec n \log\log n$. Taking logs again, and by the rule of sum, it is sufficient

[8] In the literature on L-functions, the notation "$f \preceq g$" actually means $f \leq g$ (ev.). There is a deep theory involving such functions, with connection to Nevanlinna theory.

to show $2\log\log n \prec \log n$. Taking logs again, and by the rule of sum again, it is suffices to show $\log^{(3)} n \prec \log^{(2)} n$. But the latter follows from the rule of logarithms.

EXERCISES

Exercise 13.1: (i) Simplify the following expressions: (a) $n^{1/\lg n}$, (b) $2^{2^{\lg\lg n - 1}}$, (c) $\sum_{i=0}^{k-1} 2^i$, (d) $2^{(\lg n)^2}$, (e) $4^{\lg n}$, (f) $(\sqrt{2})^{\lg n}$.
(ii) Re-do the above, replacing each occurrence of "2" (explicit or otherwise) in the previous expressions by some constant $b > 2$. Note that lg is $\log_2$, $4 = 2^2$ and $\sqrt{n} = n^{1/2}$. So when we replace these implicit 2's by c, we get $\log_b$, c^c and $n^{1/b}$ in the above expressions. ♢

Exercise 13.2: Order these in increasing big-Oh order:

$$n\lg n, \quad n^{-1}, \quad \lg n, \quad n^{\lg n}, \quad 10n + n^{3/2}, \quad \pi^n, \quad 2^n, \quad 2^{\lg n}.$$

♢

Exercise 13.3: Order the following 5 functions in order of increasing Θ-order: (a) $\log^2 n$, (b) $n/\log^4 n$, (c) $\sqrt{n}$, (d) $n2^{-n}$, (e) $\log\log n$. ♢

Exercise 13.4: Order the following functions (be sure to parse these nested exponentiations correctly): (a) $n^{(\lg n)^{\lg n}}$, (b) $(\lg n)^{n^{\lg n}}$, (c) $(\lg n)^{(\lg n)^n}$, (d) $(n/\lg n)^{n^{n/(\lg n)}}$. (e) $n^{n^{(\lg n)/n}}$. ♢

Exercise 13.5: Order the following set of 36 functions in non-increasing order of growth. Between consecutive pairs of functions, insert the appropriate ordering relationship: $\preceq$, $\asymp$, $\leq$ (ev.), $=$.

	a	b	c	d	e	f
1.	$\lg\lg n$	$(\lg n)^{\lg n}$	2^n	$2^{\lg n}$	$2^{\lg^* n}$	$2^{2^{n+1}}$
2.	$(1/3)^n$	$n2^n$	$n^{\lg\lg n}$	e^n	$n^{1/\lg n}$	$\lceil\lg n\rceil!$
3.	$2^{\sqrt{2\lg n}}$	$(3/2)^n$	2	$\lg(n!)$	n	$\sqrt{\lg n}$
4.	$2^{(\lg n)^2}$	2^{2^n}	n^2	$n\lg n$	$(n+1)!$	$4^{\lg n}$
5.	$\lg(\lg^* n)$	$\lg^2 n$	$(1+\frac{1}{n})^n$	$n^{\lg n}$	$n!$	$2^{(\lg n)/n}$
6.	$(\sqrt{2})^{\lg n}$	$\lg^* n$	$(n/\lg n)^2$	$\sqrt{n})$	$\lg^*(\lg n)$	$1/n$

NOTE: to organize of this large list of functions, we ask that you first order each row. Then the rows are merged in pairs. Finally, perform a 3-way merge of the 3 lists. Show the intermediate lists of your computation (it allows us to visually verify your work). ♢

Exercise 13.6: Order the following functions:

$$n, \quad \lceil\lg n\rceil!, \quad \lceil\lg\lg n\rceil!, \quad n^{\lceil\lg\lg n\rceil!}, \quad 2^{\lg^* n}, \quad \lg^*(2^n), \quad \lg^*(\lg n), \quad \lg(\lg^* n).$$

♢

Exercise 13.7: (Purdom-Brown) Our summation rules already gives the Θ-order of the summations below. This exercise is interested in sharper bounds:
(a) Show that $\sum_{i=1}^n i! = n![1 + \mathcal{O}(1/n)]$.
(b) $\sum_{i=1}^n 2^i \ln i = 2^{n+1}[\ln n - (1/n) + \mathcal{O}(n^{-2})]$. HINT: use $\ln i = \ln n - (i/n) + \mathcal{O}(i^2/n^2)$ for $i = 1, \ldots, n$. ♢

Exercise 13.8: (Knuth) What is the asymptotic behavior of $n^{1/n}$? of $n(n^{1/n} - 1)$?
HINT: take logs. Alternatively, expand $\prod_{i=1}^{n} e^{1/(in)}$. ♢

Exercise 13.9: Estimate the growth behavior of the solution to this recurrence: $T(n) = T(n/2)^2 + 1$. ♢

END EXERCISES

§A. APPENDIX: Exponential and Logarithm Functions

Next to the polynomials, the two most important functions in algorithmics are the **exponential function** and its inverse, the **logarithm function**. Many of our asymptotic results depend on their basic properties. For the student who wants to understand these properties, the following will guide them through some exercises. We define the **natural exponential function** to be

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

for all real x. This definition is also good for complex x, but we do not need this here. The **base of the natural logarithm** is defined to be the number

$$e := \exp(1) = \sum_{i=0}^{\infty} \frac{1}{i!} = 2.71828...$$

The next exercise derives some asymptotic properties of the exponential function.

Exercise A.1: Show that
(a) $\exp(x)$ is continuous,
(b) $\frac{d\exp(x)}{dx} = \exp(x)$ and hence $\exp(x)$ has all derivatives,
(c) $\exp(x)$ is positive, strictly increasing,
(d) $\exp(x) \to 0$ as $x \to -\infty$, $\exp(x) \to \infty$ as $x \to \infty$,
(e) $\exp(x + y) = \exp(x)\exp(y)$,

♢

We often need explicit bounds on exponential functions (not just asymptotic behavior). Derive the following bounds:

Exercise A.2:
(a) $\exp(x) \geq 1 + x$ for all $x \geq 0$ with equality iff $x = 0$.
(b) $\exp(x) > \frac{x^{n+1}}{(n+1)!}$ for $x > 0$. Hence $\exp(x)$ grow faster than any polynomial in x.
(c) For all real $n \geq 0$,

$$\left(1 + \frac{x}{n}\right)^n \leq e^x \leq \left(1 + \frac{x}{n}\right)^{n+(x/2)}.$$

It follows that an alternative definition of e^x is

$$e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n.$$

(d) $\exp(x)\left(1 - \frac{x^2}{n}\right) \leq \left(1 + \frac{x}{n}\right)^n$ for all $x, n \in \mathbb{R}$, $n \geq 1$ and $|x| \leq n$. See [15]. ♢

The **natural logarithm** function $\ln(x)$ is the inverse of $\exp(x)$: $\ln(x)$ is defined[9] to be the real number y such that $\exp(y) = x$. Note that this is a partial function because it is defined for all and only positive x.

Exercise A.3: Show that
(a) $\frac{d\ln(x)}{dx} = \frac{1}{x}$,
(b) $\ln(xy) = \ln(x) + \ln(y)$,
(c) $\ln(x)$ increases monotonically from $-\infty$ to $+\infty$ as x increases from 0 to $+\infty$. ♢

These two functions now allow us to define **general exponentiation** to any base b: any real α, we define

$$\exp_b(\alpha) := \exp(\alpha \ln(b)). \tag{106}$$

Usually, we write $\exp_b(\alpha)$ as b^α. Note that if $b = e$ then we obtain e^α, a familiar notation for $\exp(\alpha)$.

We see from (106) that b must be positive since $\ln(b)$ is otherwise undefined. Moreover, the case $b = 1$ is highly degenerate since b^α is identically equal to 1. It is easy to check that $(1/b)^\alpha = b^{-\alpha}$, and hence it is not necessary to explicitly consider the case $b < 1$ (since we can replace such a b by $1/b$ which would be > 1.

Once we have the definition of $exp_b(x) = b^x$, the **general logarithm** for any base b can be defined: $\log_b(x)$ is defined to be the inverse of the function $exp_b(x) = b^x$: $\log_b(x)$ is defined to be the y such that $b^y = x$. Note that for $b > 1$, $\log_b(x)$ is well-defined for all $x > 0$. But for $b < 1$, $\log_b(x)$ is undefined for $x > 1$. This gives another reason for avoiding bases $b < 1$. *Unless otherwise noted, the base b of our general logarithm and exponentiation is assumed to satisfy $b > 1$.*

So b^a and $\log_b a$ are derived from the special cases, e^a and $\ln a$!

Exercise A.4: We show some familiar properties: the base b is omitted if it does not affect the stated property.
(a) The most basic properties are the following two:

$$\log(ab) = (\log a) + (\log b), \quad \log_b x = (\log_c x)/(\log_c b).$$

(b) $\log 1 = 0, \quad \log_b b = 1, \quad y = x^{\log_x y}, \quad \log(x^y) = y \log x.$
(c) $\log(1/x) = -\log x, \quad \log_b x = 1/(\log_x b), \quad a^{\log b} = b^{\log a}.$
(d) $\frac{dx}{dx}(x^\alpha) = \alpha x^{\alpha-1}$.
(e) For $b > 1$, the function $\log_b(x)$ increases monotonically from $-\infty$ to $+\infty$ as x increases from 0 to ∞. At the same time, for $0 < b < 1$, $\log_b(x)$ decreases monotonically from $+\infty$ to $-\infty$. ♢

Notations for Logarithms. The Computer Science Logarithm is $\log_2$, which is denoted by lg. Authors often use $Log := \log_{10}$ for logarithm to base 10, another important base especially in engineering and finance. Our default assumption is that the base of logarithms is some $b > 1$. When the actual value of b is immaterial, we simple write 'log' without specifying the base.We also write $\log^{(k)} x$ for the k-fold application of the logarithm function to x. Thus $\log^{(2)} x = \log\log x$, and by definition, $\log^{(0)} x = x$. This is to be distinguished from "$\log^k n$" which equals $(\log n)^k$. On the black board, it is convenient to write $\ell\ell\text{og}\, n$, $\ell\ell\ell\text{og}\, n$ for $\log\log n$, $\log\log\log n$, etc.

$\log^{(k)} x$ vs. $\log^k x$

[9] This real value y is called the principal value of the logarithm. That is because if we view $\exp(\cdot)$ as a complex function, then $\ln(x)$ is a multivalued function that takes all values of the form $y + 2n\pi$, $n \in \mathbb{Z}$.

¶41. Bounds on logarithms. For approximations involving logarithms, it is useful to recall a fundamental series for logarithms:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = -\sum_{i=1}^{\infty} \frac{(-x)^i}{i} \tag{107}$$

valid for $|x| < 1$. From this, we obtain the useful bound: $x - x^2/2 < \ln(1+x) < x$. To see that $\ln(1+x) < x$ we must show that $R = \sum_{i=2}^{\infty}(-x)^i/i > 0$. This follows because if we pair up the terms in R we obtain

$$R = (x^2/2 - x^3/3) + (x^4/4 - x^5/5) + \cdots,$$

which is clearly a sum of positive terms. A similar argument shows $\ln(1+x) > x - x^2/2$.

The formula (107) allows us to compute $\ln(y)$ for any $y \in (0,2)$. How do we evaluate $\ln(y)$ for $y \geq 2$? Assume that we have good approximations to $\ln(2)$. Then we can write $y = 2^n(1+x)$ (i.e., n is the number of times we must divide y by 2 until its value is less than 2). Then we can evaluate $\ln(y)$ as $n\ln(2) + \ln(1+x)$. This procedure depends on having a good approximation to $\ln(2)$. Can we do this? Indeed,

$$\ln 2 = \sum_{k=1}^{\infty} \frac{1}{k2^k} \tag{108}$$

Using this rapidly converging series, we can quickly compute $\ln 2$ to any desired accuracy. To derive this series, note that $\frac{1}{1-x} = \sum_{i\geq 0} x^i$ and so $\int \frac{dx}{1-x} = \sum_{i\geq 0} x^{i+1}/(i+1) = \sum_{i\geq 1} x^i/i$. Putting $y = 1-x$, $\int \frac{dx}{1-x} = -\int \frac{dy}{y} = -\ln y = \ln(1/y)$. This shows $\ln \frac{1}{1-x} = \sum_{i\geq 1} x^i/i$, and (108) is just the special case where $x = 1/2$.

Mother of Series again!

Alternatively, to compute $\ln y$, we can write $y = n(1+x)$ where $n \in \mathbb{N}$ and write $\ln(y) = \ln(n) + \ln(1+x)$. To evaluate $\ln(n)$ we use the fact $\ln(n) = H_n - \gamma - (2n)^{-1} - \mathcal{O}(n^{-2})$ (see §5). Of course, this method requires approximations Euler's constant γ instead of $\ln 2$. Again, there are rapid approximations of γ.

¶42. Log-star function. We define the **log-star function**: $\log^* x$ is the maximum non-negative integer n such that $\lg^{(n)}(x)$ is defined. Thus $\log^*(x) = 0, 1, 2$ iff $x \leq 0$, $0 < x \leq 1$, $1 < x \leq 2$ (respectively). So log-star is integer-valued. Although we have used base 2 in its definition, it could be defined generally for any $b > 1$.

References

[1] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.

[2] G. Dowek. Preliminary investigations on induction over real numbers, 2003. Manuscript, http://www.lix.polytechnique.fr/ dowek/publi.html.

[3] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $o(n^{0.695})$ query time. *Info. Processing Letters*, 23:289–293, 1986.

[4] M. H. Escardó and T. Streicher. Induction and recursion on the partial real line with applications to Real PCF. *Theoretical Computer Science*, 210(1):121–157, 1999.

[5] W. Feller. *An introduction to Probability Theory and its Applications*. Wiley, New York, 2nd edition edition, 1957. (Volumes 1 and 2).

[6] M. L. Fredman. *Growth Properties of a class of recursively defined functions*. PhD thesis, Stanford University, 1972. Technical Report No. STAN-CS-72-296. PhD Thesis.

[7] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. International Computer Science Series. Addison-Wesley Publishing Company, London, 1984.

[8] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, 2nd edition, 1982.

[9] G. H. Hardy. *Orders of Infinity*. Cambridge Tracts in Mathematics and Mathematical Physics, No. 12. Reprinted by Hafner Pub. Co., New York. Cambridge University Press, 1910.

[10] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962.

[11] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Boston, 2nd edition edition, 1975.

[12] G. S. Lueker. Some techniques for solving recurrences. *Computing Surveys*, 12(4):419–436, 1980.

[13] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In *Proc. 6th Australian Software Engineering Conf. (ASWEC91)*, pages 257–270. Australian Comp. Soc., 1991.

[14] B. Mishra and A. Siegel. (Class Lecture Notes) Analysis of Algorithms, January 28, 1991.

[15] D. S. Mitrinović. *Analytic Inequalities*. Springer-Verlag, New York, 1970.

[16] J. Paul Walton Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York, 1985.

[17] R. M. Verma. A general method and a master theorem for divide-and-conquer recurrences with applications. *J. Algorithms*, 16:67–79, 1994.

[18] X. Wang and Q. Fu. A frame for general divide-and-conquer recurrences. *Info. Processing Letters*, 59:45–51, 1996.